UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

CURSO 2004-2005

OPTIMIZACIÓN DE LA TRANSFORMADA WAVELET DISCRETA (DWT)

Borja José García Menéndez
Eva Mancilla Ambrona
Ruth Montes Fraile
Tutor: Luis Piñuel Moreno

INDICE

INDICE 1
INTRODUCCIÓN 3
1. CONOCIMIENTOS PREVIOS 4
1.1 ESTÁNDAR JPEG2000 4
Características 5
Ventajas 5
Resultados 6
Áreas de aplicación del estándar JPEG 2000 8
1.2 WAVELETS 9
Compresión de datos 9
Comprensión con pérdida y sin pérdida 10
Compresión con pérdida 11
Métodos de compresión con pérdida 11
Compresión Wavelet 11
Transformada discreta wavelet (DWT) 12
Opciones del DWT en el JPEG2000 19
2.TECNOLOGÍA ALTIVEC 21
2.1. ARQUITECTURAS POWERPC 21
2.2. ALTIVEC 22
2.2.1. INTRODUCCIÓN 22
2.2.2. FUNCIONALIDADES 23
2.3. INTERFAZ DE LENGUAJES DE ALTO NIVEL (HIGH-LEVEL LANGUAJE INTERFACE) 24
2.3.1.TIPO DE DATOS 24
2.3.2. PALABRAS CLAVE 25
2.3.3. ALINEACIÓN 26
2.3.4. EXTENSIONES DE LOS OPERADORES DE C/C++ PARA LOS NUEVOS TIPOS 27
2.3.5. NUEVOS OPERADORES 27
2.3.6. INTERFAZ DE PROGRAMACIÓN 29
2.4. OPERACIONES Y PREDICADOS ALTIVEC 30
2.4.1. OPERACIONES USADAS EN EL PROYECTO DE ALTIVEC 31
3. ARQUITECTURA INTEL 39
3.1. INTRODUCCIÓN 39
Repertorio de Instrucciones SSE 39
Arquitectura Pentium III y Pentium IV 40
3.2. VECTORIZACIÓN CON INSTRUCCIONES DEL REPERTORIO SSE 41
3.3. USO DEL REPERTORIO SSE EN C 42
3.4. FUNCIONES UTILIZADAS EN NUESTRO CÓDIGO 45
4. OPTIMIZACIÓN 48
4.1 PSEUDOCODIGO PARA COLUMNAS 48
4.2 PSEUDOCODIGO PARA LAS FILAS 49
4.2.1 Vectorización del filtro 49
4.2.2 Vectorización por transposición 50

5.SPEED UP's 51
5.1 URBION TIEMPOS TOTALES 51
5.2 SPEED UP URBIÓN 52
5.3 YELMO GCC - TIEMPOS TOTALES 53
5.4 SPEED UP YELMO GCC 54
5.5 YELMO ICC TIEMPOS TOTALES 55
5.6 SPEED UP YELMO ICC 56
5.7 IMAGEN 128x128 57
5.8 SPEED UP 128x128 58
5.9 IMAGEN 4096x4096 59
5.10 SPEED UP 4096x4096 60
5.11 COMPARATIVAS FILAS-COLUMNAS 61
Referencias 64

INTRODUCCIÓN

El objetivo básico de este proyecto es optimizar la transformada directa wavelet (DWT), parte integrante del estándar JPEG2000, para las plataformas PowerPC y SSE, puesto que los algoritmos simples utilizados presentan problemas de localidad espacial y no aprovechan las operaciones que permiten la vectorización, operaciones que el amplio repertorio de ambas arquitecturas poseen.

Inicialmente comentaremos los puntos básicos necesarios que nos ayudarán a definir el ámbito de este proyecto y posteriormente nos centraremos en el trabajo realizado y en los resultados obtenidos. Para ello primeramente presentaremos el estándar JPEG2000, dando una somera definición, explicando sus múltiples usos y las mejoras que presenta frente al estándar JPEG.

A continuación nos centraremos en la DWT, parte integrante de este estándar y comienzo del desarrollo del proyecto, la definiremos y compararemos con otro tipo de técnicas de optimización con perdida y mediante algunos diagramas podremos comprender su mecanismo.

En los siguientes apartados entraremos en profundidad a comentar las optimizaciones que permiten tanto las plataformas PowerPC como las plataformas SSE. Dentro de cada uno de los apartados estudiaremos la arquitectura así como el repertorio de instrucciones del que se dispone, y más detalladamente las instrucciones necesarias que se han utilizado en las optimizaciones realizadas en la DWT.

En los siguientes apartados mostraremos parte del código generado, primeramente en una versión que minimiza en muy poco el sencillo algoritmo de la DWT pero sin usar ningún tipo de vectorización. Esta primera versión se caracteriza por almacenar la imagen no como una matriz sino como un vector en el que las columnas se almacenan continuas unas a otras en una gran fila, esto permite tener una mayor localidad espacial. Además se usan cinco punteros para acceder a la memoria y así minimizar los accesos a posiciones contiguas de memoria.

En la siguiente sección que se presenta se muestran los resultados de haber aplicado las operaciones de vectorización con el repertorio de instrucciones altivez, para la máquina con arquitectura PowerPC. Se presentan dos versiones, ambas utilizan el mismo tipo de vectorización para las columnas, pero para las filas en el primero de los casos se aplica la vectorización del filtro, y en el segundo de los casos se aplica transposición de las mismas.

A continuación se presenta la misma estructura que el punto anterior, pero esta vez utilizando el repertorio de instrucciones SSE para la arquitectura Pentium.

Para finalizar mostraremos las gráficas con los resultados obtenidos y las mejoras frente al algoritmo sin optimizar.

1. CONOCIMIENTOS PREVIOS

1.1 ESTÁNDAR JPEG2000

JPEG 2000 es un nuevo sistema de codificación basado en wavelets aplicable a diferentes tipos de imágenes estáticas (blanco y negro, niveles de gris, color, multicomponentes, etc) con diferentes características (imágenes naturales, científicas, medicas, aéreas, texto, gráficos, etc) y utilizable en diferentes escenarios (cliente / servidor, transmisión en tiempo real, archivo de imágenes, sistemas con recursos limitados de anchos de banda, etc). Este sistema de codificación permite operar a unas tasas de compresión y con una calidad subjetiva de imagen superior a anteriores estándares.

Fue creada por el comité Joint Photographic Experts Group que anteriormente había creado el algoritmo JPEG. Los objetivos técnicos eran mejorar la calidad de las imágenes tratadas y aumentar las posibles aplicaciones. Entre sus aplicaciones está la reducción de la comprensión de imágenes, reduciendo su volumen a cambio de ciertas perdidas visuales y facilitar su descarga en entornos cliente/ servidor. Estas reducciones también permiten aumentar la velocidad de descarga de las páginas Web, puesto que los gráficos e imágenes son los elementos más pesados y más lentos en la visualización de dichas páginas, y el navegador que utilicemos será el que posteriormente las descomprimirá y mostrará, aunque en la actualidad no está ampliamente soportado por los programas de visualización de páginas web. Así mismo entre sus aplicaciones esta la habilidad de manipular imágenes, con un mayor porcentaje de comprensión que el JPEG convencional, y sin provocar la aparición de “bloques” o efecto pixelazo; y también permite enviar archivos sin perdida de calidad original.

Fue la descomposición wavelet la que se adoptó como base del nuevo estándar. Actualmente la transformada wavelet está presente en innumerables e insospechados campos. A continuación podemos apreciar el diagrama de bloques del JPEG2000, constituido por una etapa de pre-tratamiento de la imagen, una transformada directa wavelet, una etapa de cuantización, una codificación aritmética (también conocida como tier-1) y después la organización de la trama (ó tier-2).

Fig. 1 – Diagrama de bloques del JPEG2000

A continuación se han clasificado las características del estándar, así como los resultados y las áreas de aplicación del estándar.

Características

- Eficiencia de compresión.
- Compresión con y sin perdidas, dependiendo de las necesidades.
- Representación con resolución múltiple.
- Decodificación SNR progresiva y en resolución.
- Fragmentación de la imagen.
- Regiones de Interés, si se desea comprimir una parte de la imagen sin perdidas.
- Protección frente a Errores.
- Acceso y procesamiento aleatorio de la trama.
- Tratamiento post-compresión.
- Posibilidad de interacciones en la codificación.
- Un formato más flexible.

Ventajas

- Mejor calidad de la imagen reconstruida para el mismo tamaño.
- Entre un 25%-35% de reducción en el tamaño de ficheros para una calidad de imagen comparable.
- Buena calidad de imagen, incluso para tasas de compresión altas, por encima de 80:1.
- Opción de baja complejidad para dispositivos con recursos limitados, como son, los teléfonos móviles, las agendas electrónicas, etc..
- Ficheros de imagen escalables, no es necesario descomprimir para reformatear la imagen. Con JPEG 2000 la imagen que mejor se ajusta al dispositivo puede ser extraída de un servidor a partir de la imagen comprimida. Como opciones se dispone de:
  - Gran variabilidad de tamaños de imagen, desde imágenes del tamaño de un icono a su tamaño completo.
  - Gran variabilidad de tipos: niveles de gris, color, etc
  - Gran variabilidad en la calidad final, desde imágenes de baja calidad hasta imágenes sin perdidas (imagen idéntica a la original).
- Transmisión progresiva a través de una estructura de ficheros en capas. Por ejemplo, a partir de un fichero de imagen de 100Kbyte de una imagen original de 512x512 píxeles, puede enviarse una versión en baja resolución de 32x32, enviando tan solo 10Kbytes. Tan solo con enviar 15Kbytes mas la resolución aumenta a 64x64. Otras capas proporcionan transmisión progresiva y visualización basado en la calidad de la imagen, componentes de color y localización espacial de la imagen.

Resultados

A continuación se presentan algunos resultados de la utilización del estándar JPEG2000, así como su comparación con el anterior estándar JPEG.

–COMPARACIÓN JPEG/JPEG2000

En la Fig. 2 se muestra una comparación entre los dos sistemas de compresión de imágenes: JPEG y JPEG 2000.

Fig. 2. Comparación de los sistemas JPEG y JPEG 2000. Imagen original (izquierda). Imagen reconstruida utilizando JPEG 2000 (80:1) (centro). Imagen reconstruida con JPEG (56:1) (derecha). Obsérvese la presencia de mayores artefactos en la imagen de la derecha frente a la del centro.

–REGIONES DE INTERÉS (ROI)

En la Fig.3 se muestra un ejemplo de la utilización de las regiones de interés (ROI) en la codificación. Sin perdidas JPEG 2000 proporciona 2.1127bpp (3.79:1). Con perdidas y ROI (ejemplo representado en la figura) se puede obtener 0.1 bpp (80:1). La imagen de la izquierda representa la imagen original. En el centro se presentan los coeficientes wavelets seleccionados por la ROI. La imagen de la derecha presenta la imagen reconstruida.

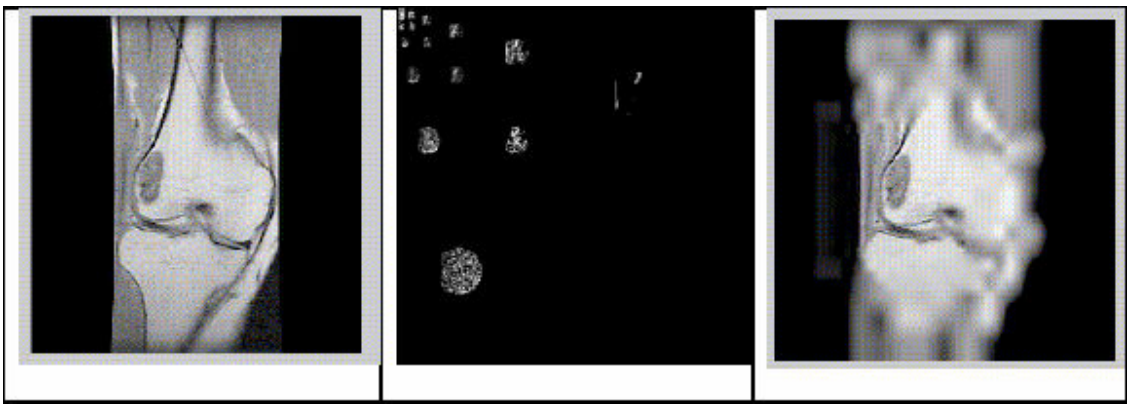


Fig. 3- Ejemplo de compresión JPEG2000 en una imagen de resonancia magnética.

–RESOLUCIÓN PROGRESIVA

En dicha modalidad se aumenta la resolución o tamaño de la imagen a medida que se va decodificando la información. Existe también otro modo denominado "SNR progresivo" en donde lo que va aumentando es la calidad de la imagen a medida que va decodificando mas información.

Fig. 4. Ejemplo de ordenación de los coeficientes wavelet denominada "resolución progresiva".

– EJEMPLOS DE COMPRESIÓN SIN PERDIDAS.

Las siguientes imágenes muestran ejemplos de compresión sin perdidas obtenidas utilizando los parámetros que facilita por defecto el sistema JPEG2000. Se presentan las imágenes originales, las reconstruidas (que son idénticas a las originales por ser sin perdidas) y los coeficientes wavelet a su lado.

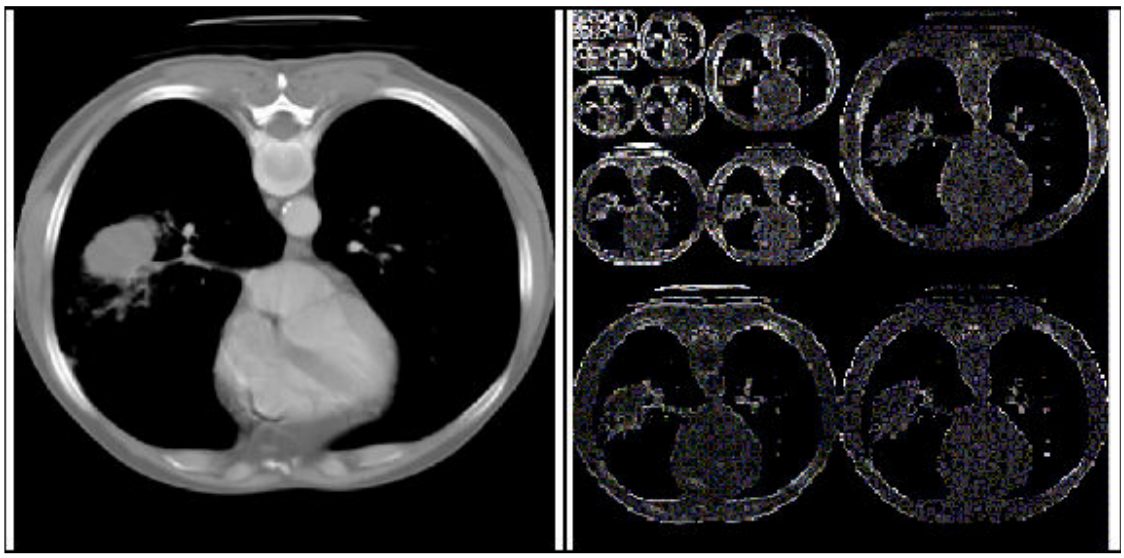


Fig. 5. Ejemplo de compresión sin perdidas mediante JPEG 2000 de una imagen de tomografía de 512x512 píxeles a 1.6584 bpp (4.82:1).

Áreas de aplicación del estándar JPEG 2000

En resumen cabe concluir que el estándar JPEG 2000 va a tener un elevado grado de impacto en la industria, y dada la sofisticación de las funciones implementadas cabe prever que tenga un periodo de vigencia mayor que el estándar anterior. En cuanto a las aplicaciones en las que JPEG 2000 va tener mayor grado de impacto, podemos citar las siguientes (la lista no es exhaustiva, pero da idea del amplio rango de aplicaciones).

–Cámaras digitales fotográficas de resolución Megapixel.
–Edición/registro de video digital (cine digital).
–Cámaras de video digital de circuito cerrado de TV para vigilancia remota o aplicaciones de seguridad.
–Cámaras Web con calidad SVGA
–Aplicaciones de imágenes en telefonía móvil de tercera (3G) y cuarta generación (4G)
–Estaciones de radiodifusión hasta frecuencias de TV estándar
–Sistemas de videoconferencia
–Transmisión sin hilos en agendas electrónicas (PDA)
–Sistemas de distribución de video e imágenes (mediante hilos o inalámbricos)
–Aplicaciones de imágenes por satélite, entre otras.

Una vez aprobado el estándar, ya empiezan a aparecer los primeros circuitos integrados que son totalmente compatibles con el estándar, como el chip CS6510 de Amphion. Para mayor información se puede consultar la página Web del comité de expertos JPEG 2000 es: www.jpeg.org.

1.2 WAVELETS

Toda transformada wavelet, también llamadas ondículas u ondeletas, considera una función, que se supone en función del tiempo, en términos de oscilaciones tanto en el tiempo como en la frecuencia. Las transformaciones wavelet se clasifican en transformadas wavelet discretas y transformadas wavelet continuas.

En cuanto a sus aplicaciones, la transformada wavelet discreta se utiliza para la codificación de señales, mientras la continua se utiliza en el análisis de señales. Como consecuencia, la versión discreta de este tipo de transformada se utiliza fundamentalmente en ingeniería e informática, mientras que la continua se utiliza sobre todo en la investigación científica.

Este tipo de transformadas están siendo cada vez más empleadas en un amplio campo de especialidades, a menudo sustituyendo a la transformada de Fourier. Se puede observar este desplazamiento en el paradigma en múltiples ramas de la física, como son, entre otros, el procesamiento de señal en general, el reconocimiento de voz, los gráficos por ordenador, pero el que más nos interesa es el proceso de imágenes.

Compresión de datos

La compresión de datos se define como el proceso de reducir la cantidad de datos necesarios para representar eficazmente una información, es decir, la eliminación de datos redundantes. En el caso de las imágenes, existen tres maneras de reducir el número de datos redundantes: eliminar código redundante, eliminar píxeles redundantes y eliminar redundancia visual.

- Código redundante: El código de una imagen representa el cuerpo de la información mediante un conjunto de símbolos. La eliminación del código redundante consiste en utilizar el menor número de símbolos para representar la información.

Las técnicas de compresión por codificación de Huffman y codificación aritmética utilizan cálculos estadísticos para lograr eliminar este tipo de redundancia y reducir la ocupación original de los datos.

- Píxeles redundantes: La mayoría de las imágenes presentan semejanzas o correlaciones entre sus píxeles. Estas correlaciones se deben a la existencia de estructuras similares en las imágenes, puesto que no son completamente aleatorias. De esta manera, el valor de un píxel puede emplearse para predecir el de sus vecinos.

Las técnicas de compresión Lempel-Ziv implementan algoritmos basados en sustituciones para lograr la eliminación de esta redundancia.

- Redundancia visual: El ojo humano responde con diferente sensibilidad a la información visual que recibe. La información a la que es menos sensible se puede descartar sin afectar a la percepción de la imagen. Se suprime así lo que se conoce como redundancia visual.

La eliminación de la redundancia esta relacionada con la cuantificación de la información, lo que conlleva una pérdida de información irreversible. Técnicas de compresión como JPEG, EZW o SPIHT hacen uso de la cuantificación.

Comprensión con pérdida y sin pérdida

Los métodos de compresión se pueden agrupar en dos grandes clases: métodos de compresión sin pérdida de información y métodos con pérdida de información. Entre las diferencias entre compresión con y sin pérdida hemos de destacar que:

- un algoritmo de compresión con pérdida puede eliminar datos para reducir aun más el tamaño, por tanto se pierde calidad, y es imposible una reconstrucción exacta de los datos originales. Los métodos de compresión con pérdida de información (*lossy*) logran alcanzar unas tasas de compresión más elevadas a costa de sufrir una pérdida de información sobre la imagen original. Por ejemplo: JPEG, compresión fractal, EZW, SPIHT, etc. Para la compresión de imágenes se emplean métodos *lossy*, ya que se busca alcanzar una tasa de compresión considerable, pero que se adapte a la calidad deseada que la aplicación exige sobre la imagen objeto de compresión.

- en el caso de la compresión sin perdida una mayor compresión solo implica más tiempo de proceso, pero los datos antes y después de la comprensión son exactamente iguales. Los métodos de compresión sin pérdida de información (*lossless*) se caracterizan porque la tasa de compresión que proporcionan está limitada por la entropía, redundancia de datos, de la señal original. Entre estas técnicas destacan las que emplean métodos estadísticos, basados en la teoría de Shannon, que permite la compresión sin pérdida. Por ejemplo: codificación de Huffman, codificación aritmética y Lempel-Ziv. Son métodos idóneos para la compresión dura de archivos.

Por tanto la compresión con pérdida solo es útil cuando la reconstrucción exacta no es indispensable para que la información tenga sentido. La información reconstruida es solo una aproximación de la información original. Por tanto suele restringirse a información analógica que ha sido digitalizada (imágenes, audio, video, etc...), donde la información puede ser parecida, y al mismo tiempo, ser subjetivamente la misma. Su mayor ventaja reside en las altas razones de compresión que ofrece en contraposición a un algoritmo de compresión sin pérdida.

Compresión con pérdida

Existen dos técnicas comúnes de compresión con pérdida:

- Por códecs de transformación: Los datos originales son transformados de tal forma que se simplifican sin posibilidad de regreso a los datos originales. Creando un nuevo conjunto de datos proclives a altas razones de compresión sin pérdida.

- Por códecs predictivos: Los datos originales son analizados para predecir el comportamiento de los mismos. Después se compara esta predicción con la realidad, codificando el error y la información necesaria para la reconstrucción. Nuevamente, el error es proclive a altas razones de compresión sin pérdida

En algunos casos se utilizan ambas, aplicando la transformación al resultado de la codificación predictiva.

Métodos de compresión con pérdida

Los distintos métodos de compresión con perdida son:
- Compresión Fractal
- JPEG
- Compresión Wavelet
- DjVu

Compresión Wavelet

Las técnicas de análisis wavelet emplean regiones de tamaño variable, para el análisis de las señales deja usar durante largo tiempo intervalos donde se necesita mucha información que precisa poca frecuencia y pequeñas regiones donde la información necesita altas frecuencias. El análisis wavelet es capaz de mostrar aspectos de la señal que otras técnicas no logran encontrar.

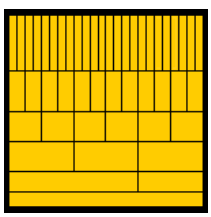


Fig. 6 - Esquema del análisis

La transformada wavelet consiste en comparar la señal con ciertas funciones wavelet, las cuales se obtienen a partir de las wavelet madre. La comparación permite obtener unos coeficientes que son susceptibles de interpretación y posterior manipulación. En cualquier caso, una característica básica es la posibilidad de invertir la transformada, recuperando la señal a partir de esos coeficientes wavelet calculados.

Transformada discreta wavelet (DWT)

El cálculo de la transformada wavelet para todas las posibles escalas supone una gran cantidad de información. Escoger solo aquellas escalas y posiciones que resulten interesantes para ciertos estudios es una tarea difícil. Si se escogen aquellas escalas y posiciones basadas en potencias de dos, los resultados serán más eficaces. Este análisis se denomina DWT .

Para muchas señales la información más importante se encuentra en las frecuencias bajas, mientras que en las altas frecuencias se encuentran los detalles o matices de la señal. El análisis wavelet permite descomponer la señal en aproximaciones y detalles, a éste proceso se le conoce con el nombre de análisis. Este filtrado nos proporciona el doble de datos de los que son necesarios, este problema se soluciona con la operación de downsampling.

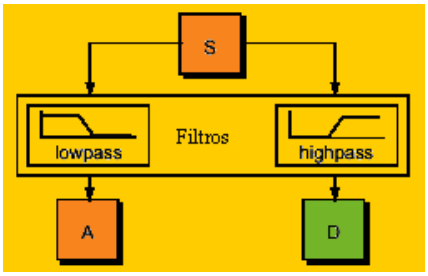


Fig. 7 - Proceso de descomposición (análisis).

El proceso de reconstrucción, también denominado síntesis, se encarga de la obtención de la señal a partir de los detalles y aproximaciones. Éste proceso se lleva a cabo con la transformada wavelet discreta inversa.

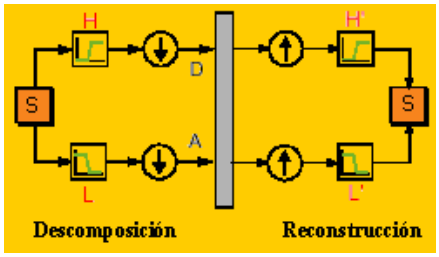


Fig. 8 - Proceso de descomposición y reconstrucción.

La elección de los filtros (wavelets) influye notablemente en los resultados finales.

La transformada directa wavelet (DWT) es muy similar a la transformada discreta de Fourier (DFT), pero en lugar de usar funciones seno y coseno como esta última, utiliza otro tipo de funciones denominadas funciones de escala y wavelets. Estas funciones reúnen la doble característica de ortogonalidad (para que la reconstrucción sea igual que la transformación), así como soporte compacto en el espacio.

La DWT aplicada a imágenes proporciona una matriz de coeficientes, conocidos como coeficientes wavelet. Si a una imagen le aplicamos la DWT obtenemos cuatro tipos de coeficientes: aproximaciones, detalles horizontales, detalles verticales y detalles diagonales. La aproximación contiene la mayor parte de la energía de la imagen, es decir, la información más importante, mientras que los detalles tienen valores próximos a cero.

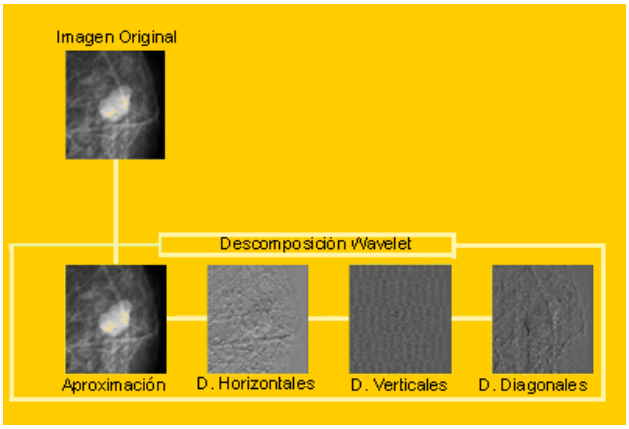


Fig. 9 -Descomposición wavelet de primer nivel.

La elección de las wavelets analizadoras juega un papel muy importante en los resultados finales. Entre las características más importantes a tener en cuenta se encuentran: soporte compacto, simetría, etc. Las wavelets biortogonales son las más eficientes para un posterior proceso de compresión, y en particular, aquellas con pocos coeficientes, ya que el coste de obtención de los coeficientes se incrementa con su número.

Generalmente, la energía de las imágenes se concentra en las frecuencias bajas. Una imagen tiene un espectro que se reduce con el incremento de las frecuencias. Estas propiedades de las imágenes quedan reflejadas en la transformada wavelet discreta de la imagen. Los niveles más bajos de compresión se corresponden con las bandas de alta frecuencia. En particular, el primer nivel representa la banda de más alta frecuencia y el nivel más fino de resolución. A la inversa, el último nivel (n) de descomposición corresponde con la banda de frecuencia más baja y la resolución más tosca. Así, al desplazarse de los niveles más altos a los más bajos, o sea, de baja resolución a alta resolución, se observa una disminución de la energía contenida en las subbandas recorridas.

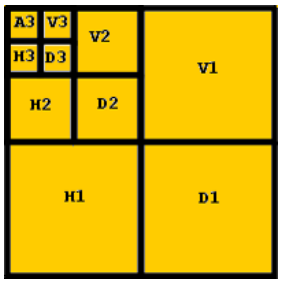


Fig. 10 - Esquema de la organización de los coeficientes wavelet.

Si los coeficientes wavelet obtenidos por medio de la transformada wavelet discreta (DWT) para un nivel concreto poseen pequeñas magnitudes (valores próximos a cero), se espera que esos coeficientes wavelet estén en los primeros niveles de descomposición. El aumento del nivel de descomposición wavelet produce unos coeficientes con mayores magnitudes. Adicionalmente, se puede comprobar como existen similitudes espaciales a través de las subbandas.

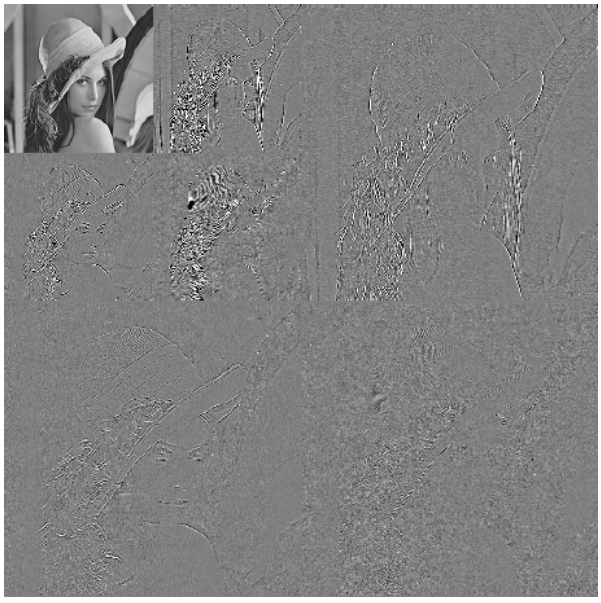


Fig. 11 - Lena

En la figura anterior se puede observar los contornos de Lena en los distintos niveles y cómo son más bastos en el primer nivel de descomposición, además de cierta similitud entre los distintos niveles.

El ejemplo más sencillo de transformada wavelet es la transformada de Haar. En este sencillo caso, un análisis (ó convolución) con las funciones base de Haar, consistentes en el calculo de la media y la diferencia entre cada dos píxeles vecinos, y dichos resultados son submuestreados de manera recursiva. Pero el estándar JPEG2000 utiliza otro tipo de funciones denominadas wavelets de Daubechies que proporcionan mejor resultado que la simple transformación de Haar[1,5].

Hay 2 procedimientos para representar el DWT que llevan a resultados idénticos: **Convolution**, y **Lifting**.

A. Convolución

La DWT descompone una secuencia 1-D (e.g., una línea de una imagen) en dos secuencias (llamadas sub-bandas), cada una con la mitad de muestras, de acuerdo al siguiente procedimiento:

1.- La sequencia 1-D es separada por un filtro en low-pass y high-pass.

2.- Las señales son comprimidas por un factor de dos para conformar las bandas low-pass y high-pass.

3.- Los dos filtros son llamados **análisis filter-bank**.

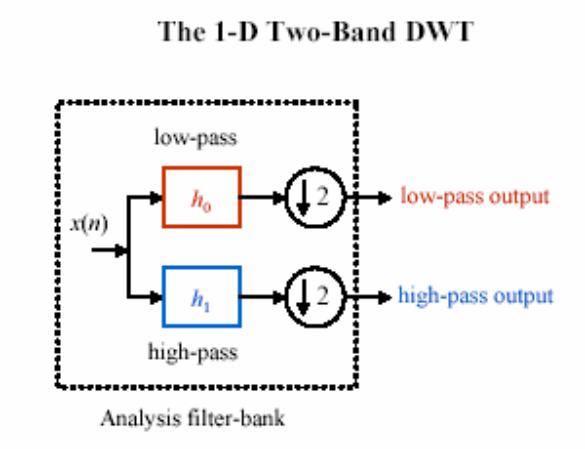


Fig. 12 – Análisis filter-bank.

En la figura 13 podremos observar un ejemplo del análisis filter-bank. Como podemos observar se aplica primero el filtro low-pass y luego el high-pass de manera sobre todos los datos. De todos los datos calculados solo nos vamos quedando con uno de cada dos datos de ambas subbandas tras realizar el downsampling, comenzando de distinta manera en cada una de las subbandas.

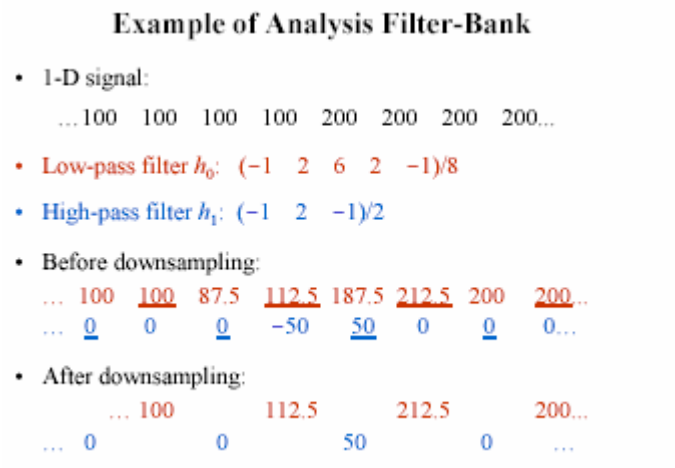


Fig. 13 – Ejemplo de Análisis filter-bank

Para calcular la DWT inversa, cada subbanda es interpolada por un factor de dos, insertando ceros entre las dos muestras y después filtrando cada secuencia resultante con los correspondientes low-pass, $g0$, o high-pass, $g1$, **sintesis filter-bank**.

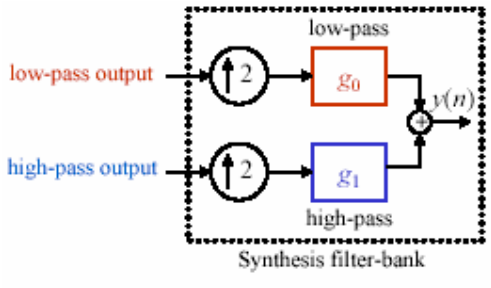


Fig.14 – Síntesis filter-bank

En la figura 15 observamos un ejemplo en el que se realiza la inversa de la DWT, se insertan ceros entre los datos de cada subbanda y se calcula la suma de las subbandas resultantes de aplicar los síntesis filter-bank.

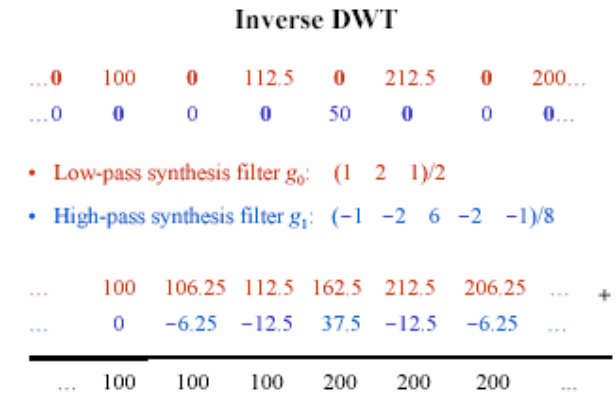


Fig. 15 – Ejemplo de Síntesis filter-bank.

Idealmente, es deseado elegir el análisis filter-bank (h0 y h1), y el síntesis filter-bank (g0 y g1), de manera que toda la distorsión del conjunto sea cero, i.e., x(n) = y(n). A esto se le llama la propiedad de reconstrucción perfecta.

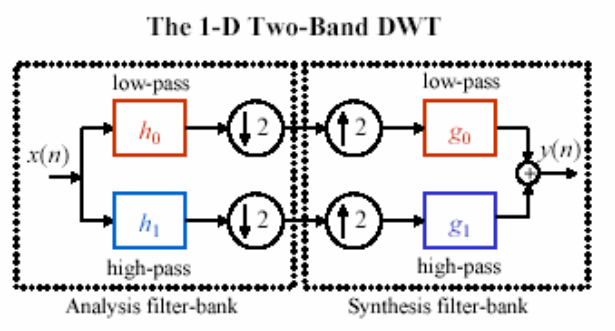


Fig. 16 – Análisis y síntesis.

La transformada discreta wavelet 2-D es igual que la DWT 1-D, pero aplicada tanto a filas como a columnas, por tanto nos queda una figura igual a la fig. 10, es decir con los datos ordenados en: aproximaciones, detalles horizontales, detalles verticales y detalles diagonales. Para visualizarlo tenemos la fig. 17 en el que se explica la estructuración de una imágen comprimida tres veces con este wavelet.

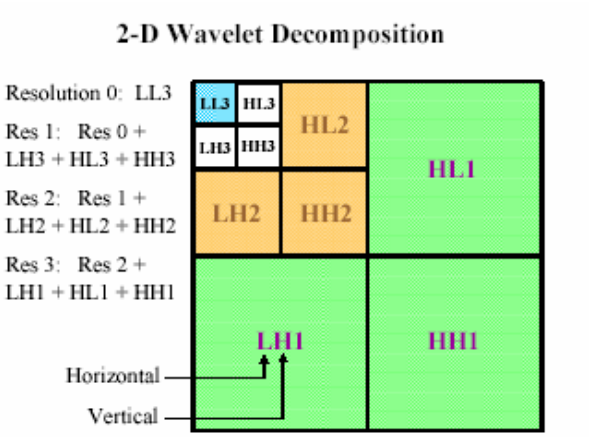


Fig. 17 – Esquema de descomposición 2-D

Para que la wavelet sea bi-ortogonal la característica que se ha de dar es que h0 sea ortogonal a g1 y que h1 sea ortogonal a g0, en el esquema de la figura 16.

B. Lifting

El esquema **lifting** es un método alternativo para procesar los coeficientes wavelet, con las siguientes ventajas:

- A veces se requiere menos procesamiento, actualmente esto depende del filter-bank específico.

- Requiere menos memoria y los coeficientes wavelet pueden ser calculados in-situ.
- Puede ser fácilmente adaptado para producir conversiones wavelet entero-entero para una compresión "lossless" (sin perdida).
- La transformada inversa es fácil de obtener y tiene la misma dificultad que la conversión directa.
- No requiere extensiones de la señal explícitas en los límites.

Su algoritmo se puede describir en 3 pasos:

1.- Split step: Primero , la señal original, *xk* , es dividida en dos secuncias pares e impares. A veces se le llama *lazy wavelet transform*:

$$s_i^0 \to x_{2i}, \qquad d_i^0 \to x_{2i+1},$$

2.- Lifting step: Este paso es ejecutado como N sub-pasos, dónde las secuencias pares e impares son convertidas usando coeficientes de predicción y actualización *Pn*(*k*) y *Un*(*k*), dichos valores dependen del filtro wavelet que esté en uso. Las relaciones son :

$$d_i^n = d_i^{n-1} + \sum P_n(k) \times s_k^{n-1}, \qquad n \in [1,2,\cdots,N]$$
$$s_i^n = s_i^{n-1} + \sum U_n(k) \times d_k^n, \qquad n \in [1,2,\cdots,N]$$

3.- Normalization step: Finalmente, los factores de normalización son aplicados para obtener los coeficientes wavelet.:

$$s_i^N \to K_0\, s_i^N, \qquad d_i^N \to K_1\, d_i^N.$$

Donde *si exp N y di exp N son*, respectivamente, los coeficientes wavelet filtro low-pass y los coeficientes fitro high-pass y *K0 y K1* son los factores de normalización correspondientes.

Cerca de los limites de la imagen , en cada sumador del entramado dónde una entrada desde la izq., la imagen reflejada de la entrada desde la dcha. es sumada dos veces. De forma análoga para el caso contrario, es decir de dcha. a izq.
El resultado es una extensión implícita de la señal limítrofe.

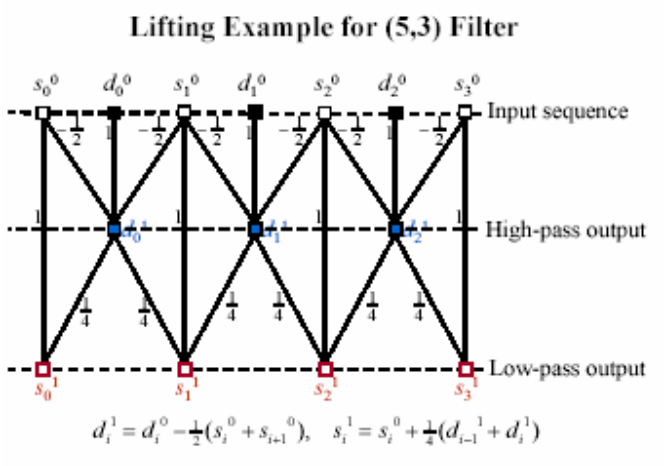


Fig. 18 – Ejemplo de lifting para el filtro (5,3)

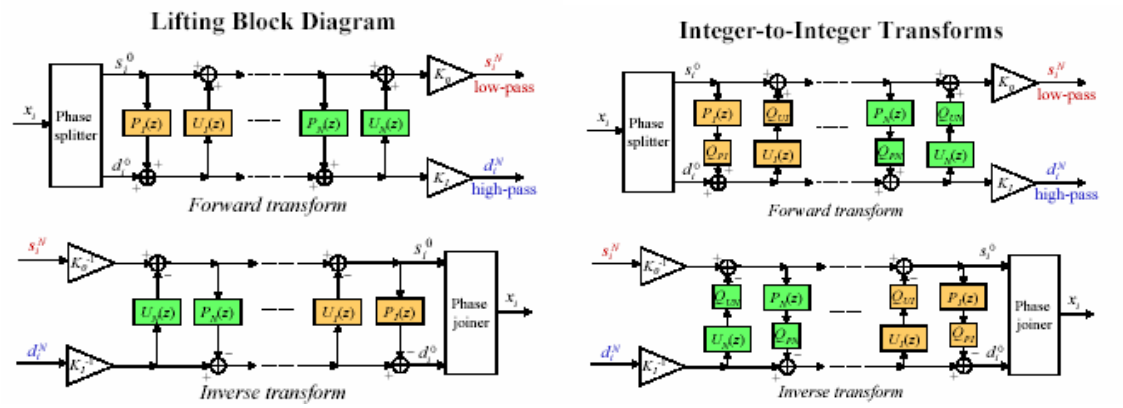


Fig. 19. Bloque de diagramas para el lifting y transformaciones entero-entero.

La estructura "lifting" puede ser fácilmente modificada para crear una entero-entero conversión para compresión "lossless" (sin perdida).

- En el análisis ,una función cuantificadora Q es añadida a la salida de cada filtro escalón.

- En el proceso de síntesis , la misma función Q se añade a la salida de cada filtro escalón.

Dos opciones redondean al entero más próximo y lo truncan.

Opciones del DWT en el JPEG2000

En JPEG2000 Parte 1, solo se permiten dos divisiones de banda LL y el uso de dos filtros DWT.

- El filtro entero (5,3) que tiene la capacidad de comprimir sin perdida y reduce la complejidad, es más rápido que el DCT, pero supeditado a alguna perdida en la codificación eficiente.

- El filtro de punto flotante Daubechies (9,7) que permite una mayor eficiencia en el código. El filtro de analisis esta normalizado a la ganancia DC de uno y a la ganancia Nyquist de dos.

La parte 2 permite filtros de tamaños arbitrarios, dependiendo de la especificación del usuario, diferentes filtros en horizontal y vertical, y arbitrarias descomposiciones de los arboles del wavelet.

2.TECNOLOGÍA ALTIVEC

2.1. ARQUITECTURAS POWERPC

Es una arquitectura de computadoras de tipo RISC creada por la Alianza AIM, un consorcio de empresas compuesto por Apple, IBM y Motorola, de cuyas primeras letras, surgió la sigla. Los procesadores de esta familia son producidos por IBM y Freescale Semiconductor que es la division de semicondutores y microprocesadores de Motorola, siendo utilizados principalmente en ordenadores o computadores Macintosh de Apple Computer.

Este microprocesador está diseñado en base a la arquitectura POWER de IBM con algunos componentes tomados del microprocesador Motorola 68000 para darle compatibilidad con arquitectura de los ordenadores de Apple.

En ella pueden ser ejecutados, al menos, los sistemas operativos:

- MacOs
- MacOSX
- AmigaOS
- FreeBSD
- GNU/Linux

Los procesadores PowerPc que existen son los siguientes:

- 601 MPC601 50 y 66 MHz
- 602 productos para consumidor (bus de datos y direcciones multiplexados)
- 603 notebooks
- 603e
- 604
- 604e
- 620 la primera implementación de 64 bits
- x704 BiCOMOS implementación PowerPC por Exponential Technologies
- 750 G3(1997) 233 MHz y 266 MHz
- 7400 G4 (1999) 350 MHz
- 750FX anunciado por IBM en 2001 y disponible en 2002 en 1 GHz.
- 970 G5 (2003) implementación 64-bit derivada del IBM POWER4 en velocidades de 1,4 GHz, 1,6 GHz, 1,8 GHz y 2,0 GHz

2.2. ALTIVEC

2.2.1. INTRODUCCIÓN

AltiVec es un conjunto de instrucciones SIMD en coma flotante y enteros diseñado y en propiedad de Apple Computer, IBM y Motorola (la alianza AIM), y puesto en ejecución en las versiones de PowerPC incluyendo el G4 de Motorola y los procesadores G5 de IBM. AltiVec es una marca registrada en propiedad de Motorola, así que el sistema también es nombrado como Motor de Velocidad por Apple y VMX por IBM.

AltiVec era el sistema SIMD más potente en una Unidad de Procesamiento Central de un ordenador de sobremesa cuando fue introducido al final de los años 90. Comparado con sus contemporáneos (MMX de sólo números enteros de Intel, SSE, en coma flotante, y los diversos sistemas de otros fabricantes RISC), AltiVec ofrecía más registros que se podían utilizar de más formas y funcionar mediante un conjunto de instrucciones mucho más flexible. Sin embargo, el sistema SIMD de cuarta generación de Intel, SSE2, introducido con el Pentium 4, tiene muchas funciones similares a las de AltiVec.

Tanto Altivec como los registros de vector de 128-bit de SSE2 pueden representar dieciséis caracteres de 8-bit con signo o sin él, ocho enteros cortos con signo o sin signo de 16-bit, cuatro enteros de 32-bit o cuatro variables en coma flotante en formato 32-bits. Ambos proporcionan instrucciones de control de la caché CPU previstas para reducir al mínimo la contaminación de la caché al trabajar con flujos de datos.

También muestran diferencias importantes. Al contrario que SSE2, AltiVec soporta un tipo de datos especial de "pixel" RGB, pero no opera con floats de doble precisión, y no hay manera de mover datos directamente entre los registros escalares y los de vectores. En armonía con el modelo de "cargar/almacentar" del diseño RISC del PowerPc, los registros del vector, como los registros escalares, sólo se pueden cargar desde la memoria, y almacenar en ella. Sin embargo, AltiVec proporciona un sistema mucho más completo de operaciones "horizontales" que trabajan a través de todos los elementos de un vector; las combinaciones permitidas de los tipos de datos y de las operaciones con éstos son mucho más completas. Se proporcionan 32 registros de vectores de 128-bit, comparado con los 8 de SSE y SSE2, y la mayoría de las instrucciones de Altivec toman tres operandos de registro, comparado con sólo dos operandos registro/registro o registro/memoria en un IA-32.

Las versiones recientes de GCC, Compilador Visual Age IBM y otros compiladores proporcionan intrínsecos para acceder a las instrucciones de AltiVec directamente desde programas en C y C++. La clase dedicada al almacenamiento "vector" aparece para permitir la declaración de los tipos nativos del vector, por ejemplo, "vector unsigned char foo;" declara una variable de 128-bit llamada "foo" que contiene dieciséis caracteres sin signo de 8-bit. Las funciones intrínsecas sobrecargadas tales como "vec_add" emiten el código op apropiado basado en el tipo de los elementos del vector, y se obliga a cumplir un fuerte tipado. En contraste, los tipos de datos definidos por Intel para los registros SIMD de IA-32 declaran solamente el tamaño del registro del vector (128 o 64 bits) y en el caso de un registro de 128-bit, si contiene números enteros o valores en la coma flotante. El programador debe seleccionar el intrínseco apropiado para los tipos de datos empleados, por ejemplo, _mm_add_epi16(x, y) para añadir dos vectores que contienen ocho números enteros de 16-bit.

Apple es el principal cliente de AltiVec, y lo usa para acelerar aplicaciones multimedia como Quicktime y ITunes y programas de procesamiento de imágenes tales como Adobe Photoshop. AltiVec también ha trabajado en partes clave del Mac OS X de Apple, incluido el compositor de gráficos Quartz. Motorola ha provisto las unidades de AltiVec en todos sus ordenadores de sobremesa desde el G4. AltiVec también se utiliza en algunos sistemas embebidos para proporcionar un proceso de la señal digital con un extremadamente alto rendimiento.

IBM ha dejado a VMX constantemente fuera de sus sistemas propietarios POWER, que están pensados para ser empleados como aplicaciones de servidor donde no es muy útil. Sin embargo, el ordenador de escritorio más reciente PowerPc 970 (apodado G5 por Apple) la CPU más reciente del tablero del escritorio de PowerPC 970 (doblado el G5 por Apple) de IBM, incluye una unidad de alto rendimiento de AltiVec. Incluye dos unidades funcionales para permitir efectos superescalares; un VMX completo en una unidad, y un multiplicador/sumador en la otra.

2.2.2. FUNCIONALIDADES

El conjunto de instrucciones de que se compone Altivec tiene como finalidad fijar una extensión a la arquitectura de PowerPc. Hay tres tipos de interfaces de programación que se pueden describir:

- Un interfaz de lenguajes de alto nivel, propuesto para el uso con lenguajes de programación tales como C ó C++.
- Una aplicación para interfaces binarios (ABI) definiendo un convenio de codificación a bajo nivel.
- Un interfaz de lenguajes de ensamblaje.

A. INTERFAZ DE LENGUAJES DE ALTO NIVEL (HIGH-LEVEL LANGUAJE INTERFACE)

El interfaz de lenguajes de alto nivel es un camino para el programador para que sea capaz de utilizar la tecnología Altivec partiendo de lenguajes de programación tales como C ó C++. Se describe un tipo de dato fundamental para el modelo de programación Altivec.

B. APLICACIÓN DE INTERFAZ BINARIO (APPLICATION BINARY INTERFACE(ABI))

El modelo de programación Altivec efectúa una extensión del PowerPC ABIs ya existente . Dicha extensión es independiente del modelo endian. ABI analiza qué tipo de datos son y qué convenios de registro están en uso para el registro de archivos de vectores. Las funciones de guardar y reestablecer en registro funciones están incluidas en la sección ABI para abogar por la uniformidad entre los compiladores en el método que se utilizar para salvar y reestablecer registros de vector.

2.3. INTERFAZ DE LENGUAJES DE ALTO NIVEL (HIGH-LEVEL LANGUAJE INTERFACE)

El interfaz para lenguajes de alto nivel Altivec se caracteriza por tener las siguientes propiedades:

- Proporciona un mecanismo eficiente y expresivo a los programadores para acceder a la funcionalidad AltiVec desde lenguajes de programación como C ó C++.

- Define un mínimo de extensiones del lenguaje. Claramente las describe con el fin de minimizar el impacto de la existencia de compiladores y herramientas de desarrollo PowerPc para el programador.

- Define las mínimas extensiones de librerías necesarias para que den soporte a la funcionalidad de AltiVec.

2.3.1.TIPO DE DATOS

El modelo de programación AltiVec introduce unos tipos de datos que son descritos a continuación en una tabla:

Nuevo tipo para C/C++	**Interpretación de contenidos**	**Valor del tipo**
vector unsigned char	16 unsigned char	0…255
vector signed char	16 signed char	-128..127
vector bool char	16 unsigned char	0(F),255(T)
vector unsigned short	8 unsigned short	0…65536
vector unsigned short int	8 unsigned short	0…65536
vector signed short	8 signed short	-32768…32767
vector signed short int	8 signed short	-32768…32767
vector bool short	8 unsigned short	0(F),65535(T)
vector bool short int	8 unsigned short	0(F),65535(T)
vector unsigned int	4 unsigned int	$0…2^{32}$-1
vector unsigned long*	4 unsigned int	$0…2^{32}$-1
vector unsigned long int*	4 unsigned int	$0…2^{32}$-1
vector signed int	4 signed int	$-2^{31}…2^{31}$ - 1
vector signed long*	4 signed int	$-2^{31}…2^{31}$ - 1
vector signed long int*	4 signed int	$-2^{31}…2^{31}$ - 1
vector bool int	4 unsigned int	0(F), 2^{32}–1 (T)
vector bool long*	4 unsigned int	0(F), 2^{32}–1 (T)

vector bool long int*	4 unsigned int	0(F), $2^{32}-1$ (T)
vector float	4 float	IEEE-754 values
vector pixel	8 unsigned short	1/5/5/5 pixel

2.3.2. PALABRAS CLAVE

El modelo introduce el uso de cinco palabras clave nuevas que sirven como identificadores. Estos son los siguientes:

- vector
- __vector
- pixel
- __pixel
- bool

Entre los tipos especificados, cuando se efectúe una declaración, el tipo *vector* debe aparecer primero. Como en los lenguajes C ó C++, la permanencia de los tipos especificados puede ser libremente entremezclada en cualquier orden, posiblemente, con otra declaración especificada. La sintaxis no permite el uso de la palabra *typedef* como tipo especificado.

Por ejemplo, lo que mostramos en el siguiente ejemplo no está permitido:

```
typedef signed short int16;
vector int16 data;
```

Estos nuevos usos pueden provocar conflictos con determinados usos en C y C++. Hay métodos que pueden ser utilizados para tratar este tipo de conflictos. Una implementación del modelo de programación de AltiVec puede elegir cualquiera de ellos.

2.3.2.1. Método de palabras clave y predefiniciones

En este método, *__vector, __pixel, y bool* son añadidos como palabras clave mientras que *vector y pixel* son predefinidos como macros. *Bool* es ya una palabra clave en C++. Esto significa que el lenguaje de C es extendido para permitir *bool* sólo como tipo especificado. Típicamente, a este tipo se le asignará a *int.* Para acomodar un conflicto con otros usos del identificador *vector* y *pixel* , el usuario puede bien utilizar *#undef* o bien usar una opción en la línea de comandos para eliminar las predefiniciones.

2.3.2.2. Método de las palabras clave sensibles al contexto

En este método, *__vector y __pixel* son añadidos como palabras clave sin mirar el contexto mientras que los nuevos usos de *vector, pixel , y bool* son palabras clave sólo en el contexto de un tipo. *vector* debe aparecer primero entre los tipos especificados. Este hecho permite que pueda ser reorganizado como un tipo especificado cuando un tipo especificado esté siendo utilizado. Los nuevos usos de *pixel y bool* se colocan después de la palabra *vector.* En todos los

otros contextos, *vector, píxel, y bool* no son palabras reservadas. Esto evita conflictos tales como *class vector, typedef int bool,* y permite la utilización de *vector, píxel, y bool* como identificadores para otros usos.

2.3.3. ALINEACIÓN

Los siguientes párrafos describen los requisitos de alineación de AltiVec. Cuando estás trabajando con vectores de datos, el programador debe ser consciente de los asuntos referidos a la alineación. Como la tecnología AltiVec no genera excepciones, el programador debe determinar cuando en el vector de datos puede dar lugar a una desalineación.

2.3.3.1. Alineación de tipos Vector

Si definimos una variable que carece de un tipo de datos concreto en memoria se considerará que siempre está alineado con una frontera de 16-byte.

Un puntero a ningún tipo de datos vector apunta a una frontera de 16-byte. El compilador es el responsable de alinear el tipo de datos vector con frontera de 16-byte. Dado que el dato vector está correctamente alineado, un programa está incorrecto si intenta desreferenciar un puntero a un tipo vector si el puntero contiene una dirección alineada con 16-byte. En la arquitectura AltiVec, una desalineación carga/almacenamiento (save/store) no causa una excepción de alineación que pueda llevar a la carga de los bytes a la dirección dada. En cambio, el bit *low-order* de la dirección es fácilmente ignorado.

2.3.3.2. Alineación de tipos no Vector

Un array de componentes para ser cargado en un vector de registros no necesita estar alineado, pero tendrá que ser accedido prestando atención a su alineación. Típicamente, esto es llevado a cabo usando operaciones como cargando el vector por desplazamiento a la derecha, vec_lvsr(), o cargando el vector con desplazamiento a la izquierda, vec_lvsl(), o la permutación del vector, vec_perm().

2.3.3.3. Alineación de tipos Vector conteniendo agregaciones y uniones

Las agregaciones, es decir, estructuras y arrays y las uniones, conteniendo tipos vector, deben estar alineadas con fronteras de 16-bytes y su organización interna acomodada, si es necesario, para que cada tipo vector interno esté alineado con una frontera de 16-bytes. Esta es una excepción para todos ABIs (AIX, Apple, SVR4, u EABI).

2.3.4. EXTENSIONES DE LOS OPERADORES DE C/C++ PARA LOS NUEVOS TIPOS

La mayoría de los operadores de C/C++ no permiten tener como argumentos los nuevos tipos de AliVec. Permite *a y b* como tipos vector y *p* como puntero a tipo vector. Los operadores normales C/C++ están extendidos para incluir las siguientes operaciones:

1.- sizeof()
Las operaciones sizeof(a) y sizeof(*p) devuelven 16.

2.- Asignación
Si en la parte derecha de la asignación o en la izquierda hay un tipo vector, entonces ambos lados de las expresión deben ser del mismo tipo vector. La expresión *a=b* es válida y representa una asignación si *a* y *b* son del mismo tipo. En otro caso, la expresión no es válida y el compilador debe mostrar un mensaje de error.

3.- Dirección del Operador
La operación *&a* es válida si *a* es un tipo vector. El resultado de la operación es un puntero a *a.*

4,. Puntero aritmético
El usual puntero aritmético puede ser interpretado como p. Particularmente, p+1 es un puntero al siguiente vector de p.

5.- Desreferenciar el puntero
Si *p* es un puntero a un tipo vector, *p implica cada carga de vector de 128-bit desde la dirección obtenida por el bit menos significativo de *p,* equivalente a la instrucción *vec_ld(0,p)* o un almacenamiento de un vector de 128-bit a esa dirección equivalente a la instrucción *vec_st(0,p).* Si es deseado para marcar el dato accedido como el menos recientemente usado (least-recently-used, LRU), la instrucción explícita vec_ldl(0,p) o vec_stl(0,p= debe ser usado.

2.3.5. NUEVOS OPERADORES

Los nuevos operadores son introducidos con el fin de construir literales de vector, ajustados a punteros, y que permite completar los accesos siempre que funcionalmente lleve a la arquitectura AltiVec.

1.- Vector literal
Un vector literal está escrito como un tipo vector entre paréntesis seguido por una expresión constante entre paréntesis. El vector literal puede ser usado como un estado de inicialización o como constantes en estados ejecutados.

La tabla que aparece a continuación nos muestra una lista de formatos y unas descripciones de literales vector. Para cada uno, el compilador genera un código que se computa o se carga esos valores en el registro.

Formato y descripción del vector literal

Notation	Represents
(vector unsigned char) (unsigned int)	A set of 16 unsigned 8-bit quantities which all have the value specified by the integer.
(vector unsigned char) (unsigned int, ..., unsigned int)	A set of 16 unsigned 8-bit quantities specified by the 16 integers.
(vector signed char) (int)	A set of 16 signed 8-bit quantities that all have the value specified by the integer.
(vector signed char) (int, ..., int)	A set of 16 signed 8-bit quantities specified by the 16 integers.
(vector unsigned short) (unsigned int)	A set of eight unsigned 16-bit quantities which all have the value specified by the unsigned integer.
(vector unsigned short) (unsigned int, ..., unsigned int)	A set of eight unsigned 16-bit quantities specified by the eight unsigned integers.
(vector signed short) (int)	A set of eight signed 16-bit quantities which all have the value specified by the integer.
(vector signed short) (int, ..., int)	A set of eight signed 16-bit quantities specified by the eight integers.
(vector unsigned int) (unsigned int)	A set of four unsigned 32-bit quantities which all have the value specified by the unsigned integer.
(vector unsigned int) (unsigned int, ..., unsigned int)	A set of four unsigned 32-bit quantities specified by the four unsigned integers.
(vector signed int) (int)	A set of four signed 32-bit quantities which all have the value specified by the integer.
(vector signed int) (int, ..., int)	A set of four signed 32-bit quantities specified by the 4 integers.
(vector float) (float)	A set of four floating-point quantities which all have the value specified by the floating-point value.
(vector float) (float, ..., float)	A set of four floating-point quantities which all have the value specified by the four floating-point values.

2.- Nueva operaciones para la representación de operadores AltiVec

Los nuevos operadores son introducidos para permitir un completo acceso a la funcionalidad proporcionada por la arquitectura AltiVec. Los nuevos operadores están representados en los lenguajes de programación por estructuras del lenguaje para parsear como las llamadas a funciones. Los nombres asociados con estas operaciones son todos prefijados con **vec_.** La representación de uno de estas formas puede indicar lo siguiente:

- Una operación AltiVec genérica, como vec_add()
- Una operación AltiVec específica, como vec_addubm()
- Un predicado computado desde una operación AltiVec como vec_all_eq()
- Carga de un vector de componentes.

Cada operador AltiVec toma una lista de argumentos que representan los operandos de entrada. El orden de los operandos es descrito en la especificación de la arquitectura y se incluye el resultado devuelto (posiblemente sea void).

El modelo de programación restringe los tipos de operandos permitidos para cada operación AltiVec, si bien es específica o genérica. El programador puede hacer caso omiso a estas obligaciones debido a que si en los argumentos efectúas un casting existe una permisibilidad de tipos.

Para una operación especificada, los tipos de los operandos determinan si la operación es aceptable sin tener en cuenta, en este caso, el modelo de programación y el tipo del resultado.
Por ejemplo, vec_vaddubm (vector signed char, vector signed char) es aceptado en el modelo de programación porque representa una alternativa razonable para hacer una adición modular con bytes de signo, mientras que vec_addubs(vector signed char, vector signed char) y vec_addubh(vector signed char, vector signed char) no son aceptables. Si fuera permitido, la primera operación produciría un resultado en el cuál la saturación trata los operandos como sin signo; la última operación produciría un resultado en que las parejas adyacentes de bytes con signo son tratadas como media palabra con signo.

Para una operación genérica, los tipos de los operandos son usados para determinar si la operación es aceptable, para seleccionar una operación en particular de acuerdo al tipo de los argumentos, y para determinar el tipo del resultado.
Por ejemplo, vec_add(vector signed char, vector signed char) puede hacer la función de vec_vaddubm() ya que puede devolver como resultado un vector de tipo char con signo (signed char) y se puede emplear en vez de vec_vadduhm() si introducimos como argumentos lo siguiente:vec_add(vector unsigned char, vector unsigned char) que devolvería como resultado un vector de char sin signo (vector unsigned char).

Las operaciones AltiVec están tratadas para que se puedan utilizar en diferentes modelos de programación. El programador no puede acceder específicamente en los nombres del registro donde está almacenado el código.

Para las instrucciones condicionales, nos devuelve como sucede en C un 1 si es true el resultado de la función y 0 si es false.

2.3.6. INTERFAZ DE PROGRAMACIÓN

No se puede acceder a la implementación del código del AltiVec pero si se añade mediante el comando #include el nombre de la librería <altivec.h> se puede acceder a las funciones que se encuentran implementadas en esa librería.

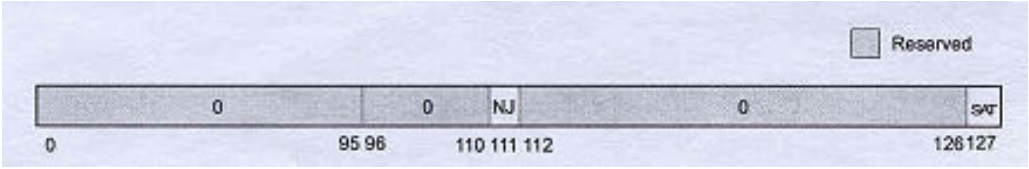
2.4. OPERACIONES Y PREDICADOS ALTIVEC

Las operaciones y predicados de que se constituye AltiVec reciben como argumentos en la mayoría de los casos el tipo vector.

Uno de los aspectos a tener en cuenta es el **vector de estados y control de registros (VSCR).** El VSCR tiene un tamaño de 32 bits, numerados de 0 a 31. Todos los bits están reservados y sin definir salvo dos:

- NJ (non-Java mode): se encuentra en el bit número 15.
- SAT (AltiVec saturation): se encuentra en el bit 31

La operación vec_mfvscr mueve el VSCR a un vector registro. Cuando se mueve, el bit 32 de VSCR es justificado a la derecha en el bit 128 del vector registro, y el bit superior 96 VRx[0-95] de vector registro está quitado, así el VSCR en un vector registro es como el de la figura:



Después de ejecutar vec_mfvscr, el resultado en el vector registro es preciso desde el punto de vista de la arquitectura. Se reflejan todas las actualizaciones en el bit SAT que podría haber sido hecho por el vector de instrucciones de manera lógica precedente en el flujo del programa.

Dentro del vector de 32 bits se desea denotar que si numeramos lo bis de 0 a 32 de izquierda a derecha, los mas a la izquierda son mas significativos que los que se encuentren más a la derecha.

En la librería **altivec.h** están implementadas numerosas funciones. En esta documentación se van a nombrar aquellas operaciones que se han utilizado para elabora la implementación que se adjunta.

2.4.1. OPERACIONES USADAS EN EL PROYECTO DE ALTIVEC

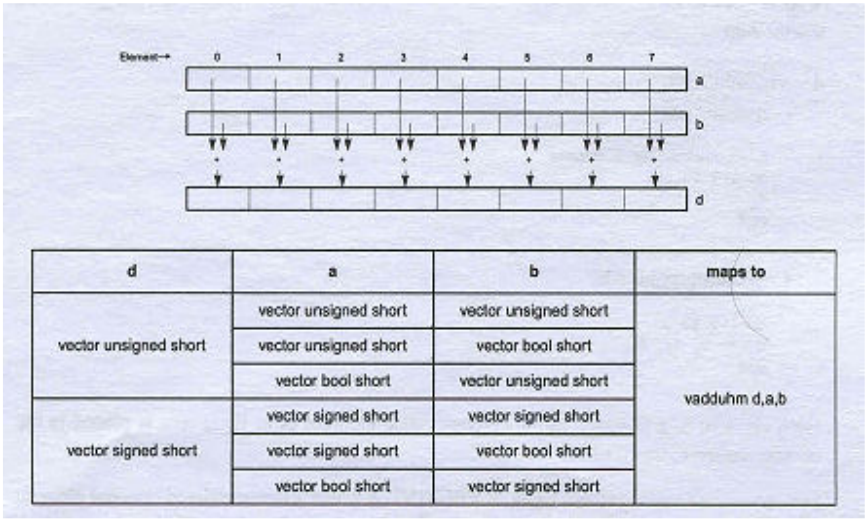
Las operaciones que se van a describir de manera exhaustiva a continuación son las que aparecen reflejadas a lo largo de nuestro código de la transformada wavelet.

Además, las operaciones han sido utilizadas con vectores de tipo entero así que serán explicadas en ese caso que es el que se ha prestado una mayor atención en la elaboración del código.

- **Operación vec_add(a,b):** consideramos a y b como de tipo vector. La función a realizar es la suma de los elementos de ambos vectores:

```
d= vec_add(a,b)
Para la suma de enteros:
   n<- number of elements
    do i=0 to n-1
    di<- ai + bi
   end
```

Cada elemento de **a** es sumado con el correspondiente elemento de **b.** Cada suma es colocada en la posición correspondiente en **d**.



d	a	b	maps to
vector unsigned short	vector unsigned short	vector unsigned short	vadduhm d,a,b
	vector unsigned short	vector bool short	
	vector bool short	vector unsigned short	
vector signed short	vector signed short	vector signed short	
	vector signed short	vector bool short	
	vector bool short	vector signed short	

Suma para 4 elementos enteros (32 Bits)

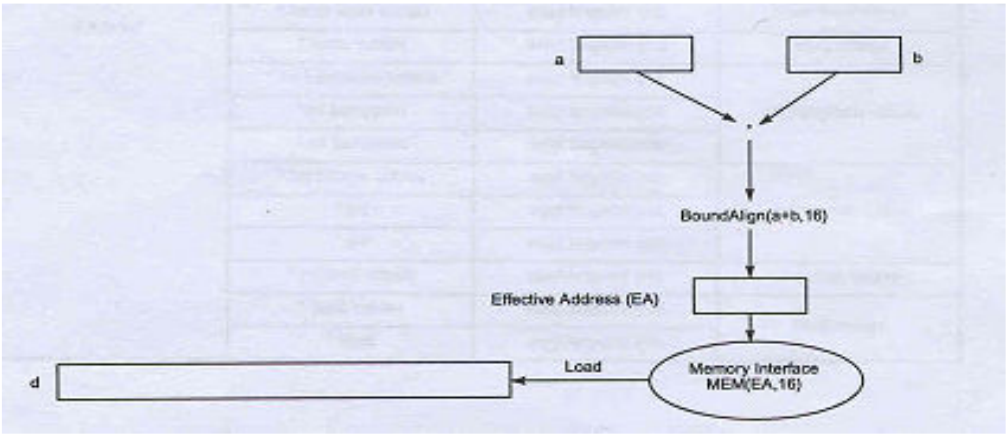
- **Operación vec_ld(a,b)**: el uso de esta operación da como salida un vector el cual lo cargamos en una variable de su mismo tipo (en nuestro ejemplo se va a llamar **d)**. En este caso, **a** va a ser considerada como el desplazamiento y **b** la dirección de la que

partimos. Se suman **a** y **b** y el vector que se encuentra en esa dirección de memoria es lo que da como resultado final esta operación.

d = vec_ld(a,b)

EA<- BoundAlign(a+b,16)
D<- MEM (EA, 16)

Una figura que representa claramente la operación que se realiza a través de d= vec_ld(a,b) es la siguiente:



La tabla que aparece a continuación nos muestra los tipos de que puede ser a y b y el posterior resultado que esto conlleva:

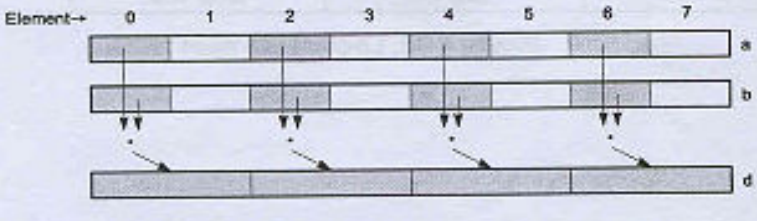
d	a	b	maps to
vector unsigned char	any integral type	vector unsigned char *	lvx d,a,b
	any integral type	unsigned char *	
vector signed char	any integral type	vector signed char *	
	any integral type	signed char *	
vector bool char	any integral type	vector bool char *	
vector unsigned short	any integral type	vector unsigned short *	
	any integral type	unsigned short *	
vector signed short	any integral type	vector signed short *	
	any integral type	short *	
vector bool short	any integral type	vector bool short *	
vector pixel	any integral type	vector pixel *	
vector unsigned int	any integral type	vector unsigned int *	
	any integral type	unsigned int*	
	any integral type	unsigned int *	
vector signed int	any integral type	vector signed int *	
	any integral type	int *	
	any integral type	int *	
vector bool int	any integral type	vector bool int *	
vector float	any integral type	vector float *	
	any integral type	float *	

- **Operación vec_mule(a,b):** Esta operación multiplica los vectores **a** y **b** y guarda el vector resultante en **d.**

```
 d = vec_mule(a,b)
n<- number of Elements in d
 do I=0 to n-1
di<- ai * bi
end
```

La función recorre los vectores a y b desde 0 al tamaño del vector a menos 1 y va calculando la multiplicación de los elementos que se encuentran en cada posición y ese valor lo almacena en un vector d.

La tabla que aparece a continuación nos muestra los tipos de que puede ser a y b y el posterior resultado que esto conlleva:



d	a	b	maps to
vector unsigned int	vector unsigned short	vector unsigned short	vmuleuh d,a,b
vector signed int	vector signed short	vector signed short	vmulesh d,a,b

- **Operación vec_perm(a,b,c):** consideramos a, b y c de tipo vector. En nuestra aplicación **a** y **b** son de vectores de enteros y **c** es un vector de unsigned char de la forma siguiente:

```
vector unsigned char m = {0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,
                          0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37};
```

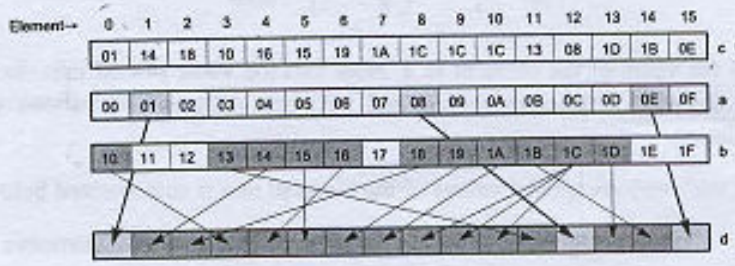
El ejemplo consiste en que como resultado en **c** van a aparecer los 8 primeros elementos del vector **a** y a continuación los 8 primeros elementos del vector **b** en ese orden.

0x20 significa que queremos en la primera posición del vector **c** la primera posición del vector **a**. Que sea el vector a nos lo dice el dos y que solicitamos la posición 0 pues el 0. Si apareciera en vez de un 2 un 3, querríamos la posición cero del vector **b**. La posición a solicitar está expresada en hexadecimal.

Por tanto, con esta función lo que se realiza es la combinación de posiciones de dos vectores y aparece como resultado en un tercer vector.

```
c= vec_perm(a,b,m);
```

Viene reflejado claramente en el diagrama que aparece a continuación:



d	a	b	c	maps to
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char	vperm d,a,b,c
vector signed char	vector signed char	vector signed char	vector unsigned char	
vector bool char	vector bool char	vector bool char	vector unsigned char	
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned char	
vector signed short	vector signed short	vector signed short	vector unsigned char	
vector bool short	vector bool short	vector bool short	vector unsigned char	
vector pixel	vector pixel	vector pixel	vector unsigned char	
vector unsigned int	vector unsigned int	vector unsigned int	vector unsigned char	
vector signed int	vector signed int	vector signed int	vector unsigned char	
vector bool int	vector bool int	vector bool int	vector unsigned char	
vector float	vector float	vector float	vector unsigned char	

La tabla que se puede observar justo arriba nos muestra de qué tipos pueden ser los argumentos y dependiendo de cuáles esos fueran, el tipo de la salida que se va a encontrar como resultado.

- **Operación vec_sra(a,b):** sean **a** y **b** dos variables de tipo vector. Se realiza un desplazamiento a la derecha de los elementos de **a** tantas veces como nos lo indique en el elemento de esa misma posición a la que hemos accedido en **a** del vector **b**. Con ese desplazamiento se consigue una división 2^i siendo i la posición del vector de b correspondiente en cada iteración.

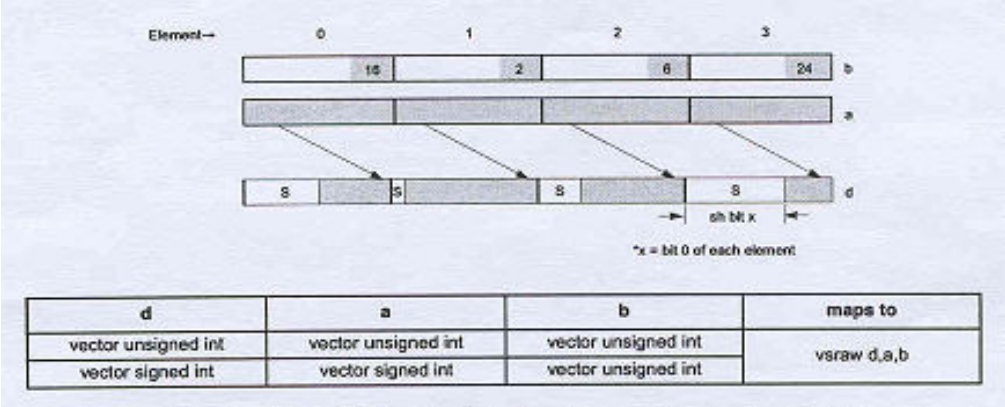
  En nuestro caso lo usamos de la siguiente manera:

```
vector int a = vec_ld(0, &(wave[2][j]));
vector unsigned int b = {1,1,1,1};
d = vec_sra(a,b);

nota: wave es una matriz
```

Cuando aplicamos vec_sra con los argumentos **a** y **b** realizamos sobre el vector a un desplazamiento en cada posición del mismo de uno. Por tanto cada posición queda dividida por 2^1=2.

Se puede observar que la explicación efectuaba queda reflejada en el diagrama siguiente:



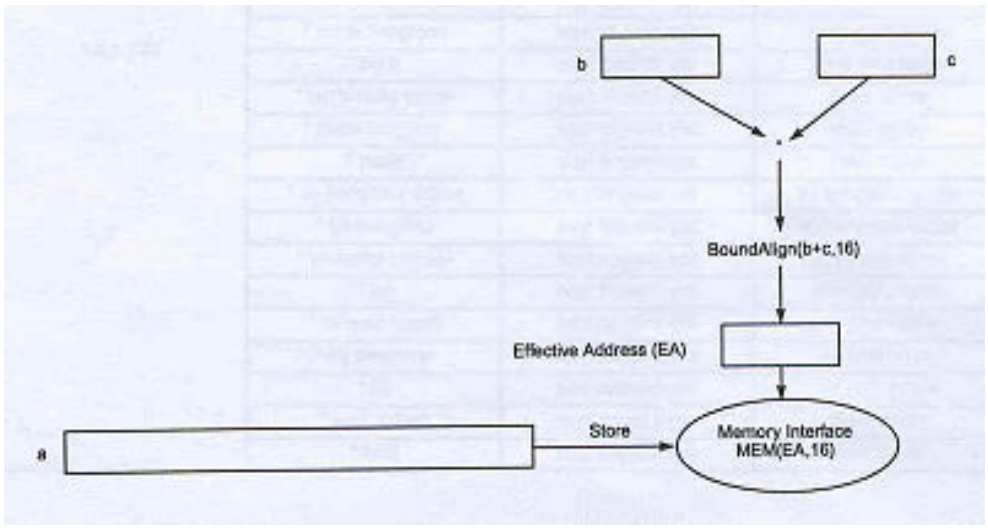
d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vsraw d,a,b
vector signed int	vector signed int	vector unsigned int	

- **Operación vec_st(a,b,c):** la función de esta operación consiste en que la variable que introducimos como primer argumento, en este caso a, es almacenada en la dirección que se obtiene como resultado de la suma entre los argumentos segundo y tercero, en este caso b y c respectivamente.

  El segundo argumento es el desplazamiento y el tercer argumento la dirección de la que partimos.

Por ejemplo: vec_st(valor_actual, 0, &(wave[0][j]));

El diagrama que aparece a continuación aclara la funcionalidad de vec_st.



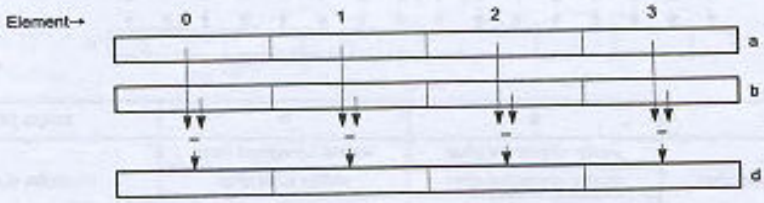
La tabla que mostramos nos define los tipos de argumentos que pueden introducirse a la operación:

a	b	c	Maps to
vector unsigned char	any integral type	vector unsigned char *	stvx a,b,c
vector unsigned char	any integral type	unsigned char *	
vector signed char	any integral type	vector signed char *	
vector signed char	any integral type	signed char *	
vector bool char	any integral type	vector bool char *	
vector bool char	any integral type	unsigned char *	
vector bool char	any integral type	signed char *	
vector unsigned short	any integral type	vector unsigned short *	
vector unsigned short	any integral type	unsigned short *	
vector signed short	any integral type	vector signed short *	
vector signed short	any integral type	short *	
vector bool short	any integral type	vector bool short *	
vector bool short	any integral type	unsigned short *	
vector bool short	any integral type	short *	
vector pixel	any integral type	vector pixel short *	
vector pixel	any integral type	unsigned short *	
vector pixel	any integral type	short *	
vector unsigned int	any integral type	vector unsigned int *	
vector unsigned int	any integral type	unsigned int *	
vector signed int	any integral type	vector signed int *	
vector signed int	any integral type	int *	
vector bool int	any integral type	vector bool int *	
vector bool int	any integral type	unsigned int *	
vector bool int	any integral type	int *	
vector float	any integral type	vector float *	
vector float	any integral type	float *	

- **Operación vec_sub(a,b):** consideramos **a** y **b** de tipo vector. La función de esta operación consiste en que se van recorriendo las posiciones de ambos vectores y para cada posición se restan los elementos y se almacena el resultado en un tercer vector al que denominamos en este caso **d**.

```
d= vec_sub (a,b)
n<- number of elements
do I=0 to n-1
di <- ai – bi
end
```

La tabla nos muestra los elementos que se pueden introducir como argumentos y dependiendo de estos cuales sean su posterior tipo de resultado.



d	a	b	maps to
vector unsigned int	vector unsigned int	vector unsigned int	vsubuwm d,a,b
	vector unsigned int	vector bool int	
	vector bool int	vector unsigned int	
vector signed int	vector signed int	vector signed int	
	vector signed int	vector bool int	
	vector bool int	vector signed int	

3. ARQUITECTURA INTEL

3.1. INTRODUCCIÓN

Repertorio de Instrucciones SSE

La SSE (*Streaming SIMD Extensions*) es un repertorio de instrucciones SIMD (*Single Instruction Multiple Data*) diseñado por Intel e introducido en sus procesadores Pentium III como respuesta al 3DNow! de los AMD que salió un año antes.

Originalmente se conocía a este repertorio como KNI, de *Katnai New Instructions* (Katmai es el nombre del proyecto de los Pentium III). A lo largo del proyecto Katmai Intel intentaba distinguir este de sus antecesores, sobre todo de su buque insignia, el Pentium II. Al final AMD añadió soporte para las instrucciones SSE en su procesador Athlon XP.

Intel por lo general no estaba satisfecho con su primera campaña de IA-32 SIMD, MMX. Dicho MMX tenia dos problemas principales: Volvía a usar existentes registros en punto flotante haciendo a la CPU incapaz de trabajar en punto flotante y datos SIMD a la vez, y solo funcionaba para enteros.

SSE añadió ocho nuevos registros de 128 bits conocidos desde XMM0 hasta XMM7. Cada registro contenía a la vez cuatro números en punto flotante de 32 bits.

Como esos registros eran adicionales, el sistema operativo debía protegerlos de los cambios de tarea, estaban desactivados por defecto hasta que el sistema operativo los habilitaba de una manera explicita. Esto quiere decir que el sistema operativo debía saber como usar las instrucciones FXSAVE y FXRSTR, que es el par de instrucciones extendidas que pueden salvar todos los estados de los registros x86, MMX, 3dNow! y SSE a la vez. Este apoyo fue rápidamente añadido a los sistemas operativos IA-32 más importantes.

Porque las SSE añadía soporte para punto flotante, se le veía mucho más uso ahora que los MMX, que las tarjetas gráficas que tenían que hacer los cálculos enteros internamente. Las operaciones SIMD enteras podían trabajar todavía con los ocho registros MMX de 64 bits. Los registros MMX están “alineados” en lo más alto de los ocho registros FPU

Sin embargo, en el Pentium III, SSE está implementado usando los mismos circuitos que el FPU, significando una vez más que la CPU no puede enviar instrucciones FPU y SSE a la vez por el pipeline. Los registros separados permiten mezclar operaciones SMID y de punto flotante sin tener que cambiar explícitamente del modo MMX al punto flotante (y viceversa)

Los Pentium IV de Intel implementan las SSE2, que son una ampliación del repertorio básico de instrucciones SSE. Las características más importantes de este SSE2 vienen dadas por la doble precisión (64 bits) de números en punto flotante (el SSE básico solamente tenia precisión simple) y el soporte para datos de tipo entero en los registros de vectores de 128 bits introducidos con los SSE, permitiendo la programación para evitar los registros MMX y FPU.

Otra mejora del repertorio SSE2 es que incluye un conjunto de instrucciones para el control de la caché, intentando al principio minimizar los datos inservibles de esta memoria cuando procesamos una cantidad de información indefinida. Otra de las causas para añadir este tipo de instrucciones es que son un sofisticado complemento de las instrucciones de conversión de formato de números.

Las primeras instrucciones SSE2 fueron introducidas en la versión inicial del Pentium IV en 2001. Este repertorio de instrucciones ha sido a su vez ampliado por el SSE3, también conocido como "*Prescott New Instructions*", y fue introducido por Intel en los Pentium IV en la primera parte del año 2004

El rival de Intel en la fabricación de chips (AMD), añadió después soporte para el repertorio SSE2, con la introducción de registros de CPU de 64 bits en su Opteron y Athlon 64, en el año 2003. No obstante, AMD amplió la implementación original de Intel para el SSE2, doblando el número de registros XMM, de ocho a dieciséis, esto es, desde el XMM0 hasta el XMM15. Los registros adicionales solamente eran visibles cuando el procesador estaba ejecutando en el modo de 64 bits largo, usando la tecnología AMD64. Intel también adoptó eventualmente esta ampliación de registros XMM en el 2004 junto con la arquitectura AMD64 (también conocida como EMT64)

Arquitectura Pentium III y Pentium IV

El Pentium III es un microprocesador de arquitectura x86 (más concretamente, un i686) de la marca Intel, que vio la luz el 26 de Febrero de 1999. Las primeras versiones fueron muy similares a su antecesor, el Pentium II, siendo las diferencias más notables el añadido del repertorio de instrucciones SSE (del que hemos hablado anteriormente) y la introducción de un polémico número de serie que era insertado en el chip durante el proceso de creación.

La versión original del Pentium III (Katmai) era casi lo mismo que el Pentium II, las únicas diferencias, a parte de la introducción del repertorio SSE, era la mejora del controlador del nivel L1 de caché (que era la causa del menor rendimiento obtenido sobre los últimos Pentium II). Al principio tenían una velocidad entre 450 y 500 Mhz.

La segunda versión, Coppermine, tenia integrado un segundo nivel de caché de 256 KB con una latencia mucho menor, que le daba mayor rendimiento que a Katmai. Debido a la gran

competencia de los procesadores Athlon AMD, Intel también volvió a trabajar en el chip internamente, lo que supuso una gran mejora en el proceso de las instrucciones.

El Pentium IV es la séptima generación de la arquitectura de microprocesadores x86 fabricados por Intel y es su primer diseño de CPU totalmente nuevo, llamada arquitectura *NetBurst*, desde el Pentium Pro de 1995. El Pentium IV original, llamado "*Willamette"*, tenía una velocidad de 1'4 y 1'5 GHz y fue lanzado en Noviembre del 2000. Un notable cambio del Pentium IV fue el de un rapidísimo 400 MHz FSB; En realidad su velocidad era 100 MHz "Quad-pumped", pero su teórico ancho de banda era cuatro veces el de un bus normal, y por eso se consideraba que podía tener esos 400 MHz de velocidad (Su competidor mas rápido tenía 266 MHz, 133 MHz pero "Double-pumped").

Para sorpresa de la mayoría de los observadores de la industria, el Pentium IV no mejoraba en el viejo diseño P6 en ninguna de dos de las medidas comunes de rendimiento: Velocidad de procesamiento de enteros o rendimiento en punto flotante. En cambio, este sacrificio por ciclo ayudaba a ganar en dos cosas: Una frecuencia de reloj muy grande, y el rendimiento del repertorio SSE.

El rendimiento del Pentium IV es mucho menor si miramos el trabajo por ciclo de reloj y lo comparamos con otras CPUs (Como pueden ser varios Athlon o las viejas arquitecturas Pentium III) pero la idea original de diseño era sacrificar instrucciones por ciclo de reloj, a cambio de poder hacer muchas más instrucciones por segundo (esto es, un mayor frecuencia de reloj).

3.2. VECTORIZACIÓN CON INSTRUCCIONES DEL REPERTORIO SSE

La introducción del repertorio SSE nos permite vectorizar nuestro código realizado en lenguaje de alto nivel (en nuestro caso C). Las optimizaciones de alto nivel incluyen intercambio de bucles, fusión de bucles, desenrollado de bucles, etc...

Para los bucles de tipo entero, la tecnología MMX™ y la Streaming SIMD Extensions proporciona instrucciones SIMD para la mayoría de los operadores aritméticos y lógicos en tipos de enteros de 32 bits, 16 bits y 8 bits. Habrá que proceder a la vectorización si la precisión final de la aritmética entera se preservará. Un desplazamiento a la derecha con el operador trabajando sobre datos de 32 bits, por ejemplo, no será vectorizable si el resultado final almacenado es un resultado de 16 bits. Nótese también que debido a que los repertorios MMX™ y SSE no son totalmente ortogonales (No soportan desplazamientos de byte, por ejemplo), actualmente no todas las operaciones de enteros pueden ser vectorizadas.

Para bucles que operan con una precisión simple de 32 bits y bucles de doble precisión de 64 bits de números en punto flotante, el repertorio SSE proporciona instrucciones SIMD para los operadores aritméticos suma, resta, multiplicación y división. También proporciona instrucciones SSE para los operadores binarios MIN, MAX, y el operador unario SQRT.

Algunas instrucciones SIMD de otros operadores matemáticos (como las funciones trigonométricas seno, coseno, tangente) están soportados en software mediante una librería.

3.3. USO DEL REPERTORIO SSE EN C

Para poder utilizar las instrucciones del repertorio SSE en nuestra aplicación en C, deberemos incluir dos librerias: *emmintrin.h,* que es la librería de las principales cabeceras de ficheros de las intrínsecas SSE2, y *xmmintrin.h,* donde se encuentran las intrínsecas para las instrucciones MMX.

Los procesadores Intel Pentium IV y otros procesadores de la misma marca tienen instrucciones para permitir el desarrollo de aplicaciones optimizadas. Las instrucciones están implementadas a través de ampliaciones de las instrucciones previamente implementadas. Esta tecnología utiliza la técnica SIMD. Por el procesamiento de datos en paralelo, las aplicaciones con un flujo de bits medio-alto son capaces de mejorar su rendimiento significativamente. El procesador Intel Itanium también soporta estas instrucciones.

Las intrínsecas son una extensión especial del código que permiten usar la sintaxis de las llamadas a las funciones de C, y las variables de este lenguaje en vez de los registros hardware.

La disponibilidad de las intrínsecas en los distintos procesadores Intel, viene dada por el siguiente cuadro:

Procesadores	Tecnología MMX™ Intrínseca	SSE	SSE2	Instrucciones del procesador Itanium
Itanium	X	X	N/A	X
Pentium IV	X	X	X	N/A
Pentium III	X	X	N/A	N/A
Pentium II	X	N/A	N/A	N/A
Pentium con tecnología MMX	X	N/A	N/A	N/A
Pentium Pro	N/A	N/A	N/A	N/A
Pentium	N/A	N/A	N/A	N/A

Los mayores beneficios de usar las intrínsecas es que tenemos nuevas características que no están disponibles utilizando el código convencional. Las intrínsecas nos permiten usar la sintaxis de C en casos, en vez de tener que programar en lenguaje ensamblador. La mayoría de

las instrucciones de los repertorios MMX™, SSE y SSE2 tienen su correspondiente intrínseca correspondiente en C, que implementa cada instrucción directamente.

La tecnología MMX y las instrucciones SSE utilizan nuevas características, como son:

- Nuevos registros. Permiten empaquetar datos de hasta 128 bits para un proceso SIMD óptimo.
- Nuevos tipos. Permiten agrupar hasta 16 elementos de datos en un solo registro.

Las SSE2 están únicamente definidas para IA-32, no para los sistemas de arquitectura Itanium. SSE2 opera en cantidades de 128 bits (dos números de punto flotante de precisión doble de 64 bits). La arquitectura Itanium no soporta procesamiento paralelo de doble precisión, por lo que el repertorio SSE2 no esta implementado para estos sistemas.

Como hemos comentado antes, el repertorio SSE usa ocho registros de 128 bits (xmm0 a xmm7). Estos nuevos registros de datos permiten el proceso en paralelo, porque cada registro puede almacenar mas de un elemento de datos, el procesador puede procesar más de un elemento a la vez. Para cada correspondiente instrucción de manipulación computacional y de datos en la nueva ampliación, hay una correspondiente intrínseca en C que implementa esa instrucción directamente. Eso permite al programador librarse de manipular registros directamente y de programar en lenguaje ensamblador. Además, el compilador optimiza las instrucciones para que el ejecutable sea más rápido.

Las funciones intrínsecas usan cuatro tipos de datos nuevos de C como operandos. La siguiente tabla, muestra los tipos de datos disponibles para las distintas extensiones:

Nuevo tipo de datos	Tecnología MMX ™	SSE	SSE2	Procesador Itanium
__m64	X	X	X	X
__m128	N/A	X	X	X
__m128d	N/A	N/A	X	X
__m128i	N/A	N/A	X	X

Tipo de datos __m64: Este tipo es utilizado para representar el contenido de un registro MMX, registro usado por la tecnología del mismo nombre. Puede guardar ocho valores de 8 bits, cuatro de 16 bits, dos de 32 bits o un único valor de 64 bits.

Tipo de datos __m128: Este tipo es utilizado para representar los contenidos de un registro SSE y puede almacenar cuatro valores de punto flotante de 32 bits cada uno.

El tipo __m128d, almacenará dos valores de punto flotante de 64 bits, y el __m128i almacena valores enteros, y estos pueden ser dieciséis de 8 bits, ocho de 16 bits cuatro de 32 bits o dos de 64 bits.

Como estos tipos no están incluidos en los tipos básicos ANSI de C, hay que tener en cuenta las siguientes restricciones:

- Hay que usar estos tipos solo en ambos lados de una asignación, como un valor devuelto, o como parámetro. No se puede usar con otras expresiones aritméticas (+, -, etc)
- Usar los tipos como objetos globales, como un acceso a los bytes y estructuras
- Solamente se pueden usar con las respectivas intrínsecas que lo permitan

La mayoría de las intrínsecas usan una convención de notación, que es la siguiente:

__mm_<operación>_<sufijo>

Donde operación indica la operación básica que se realizará. Por ejemplo, add para la suma, o sub para la resta (del inglés addition y subtraction). Mientras que el sufijo denota el tipo de datos que se operan en la instrucción.

Un número añadido a un nombre de una variable, indica el elemento de un elemento del vector, por ejemplo, r0 es la palabra menos significativa de r. Algunas intrínsecas son "compuestas" porque requieren más de una instrucción para implementarlas.

Los valores de los vectores están representados en orden de derecha a izquierda, usando el valor menos significativo para las operaciones escalares. Por ejemplo, consideremos los siguientes ejemplos de operaciones:

double a[2] = {1.0, 2.0};

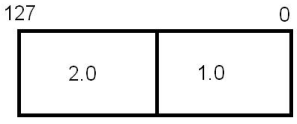
__m128d t = __mm_load_pd(a);

El resultado es el mismo que si ejecutáramos una de las dos instrucciones siguientes:

__m128d t = _mm_set_pd (2.0, 1.0);

__m128d t = _mm_setr_pd(1.0, 2.0);

Y el resultado en el registro xmm que guarda la variable t, sería el siguiente



Siendo el elemento escalar el 1.0.

3.4. FUNCIONES UTILIZADAS EN NUESTRO CÓDIGO

Para realizar nuestro código, hemos utilizado intrínsecas del repertorio SSE2. A continuación detallamos las funciones más importantes que hemos utilizado

El tipo de datos que hemos utilizado es el __m128i, donde guardaremos cuatro enteros de 32 bits. Aunque también usaremos el __m128, ya que la multiplicación no esta disponible para el tipo __m128i, con lo que al querer multiplicar dos elementos de este tipo, haremos una conversión a __m128, los multiplicaremos, y lo volveremos a convertir a _m128i. Esto no sería necesario si hubiese una función para multiplicar dos __m128i.

__m128i _mm_set_epi32(int i3, int i2, int i1, int i0)

Pone los cuatro valores enteros de 32 bits:

r0 := i0 r1 := i1

r2 := i2 r3 := i3

__m128i _mm_load_si128(__m128i const *p)

Carga 128 bits, teniendo que estar la dirección p alineada en memoria.

__m128i _mm_loadu_si128(__m128i const *p)

Carga 128 bits, no teniendo que estar la dirección p obligatoriamente alineada. Esta operación de carga es más potente que la anterior, pero también es más costosa, asi que es altamente recomendable utilizar la anterior siempre que se pueda.

__m128i _mm_add_epi32(__mm128i a, __mm128i b)

Suma los cuatro elementos contenidos en a, con los de la misma posición en b.

__m128i _mm_sub_epi32(__mm128i a, __mm128i b)

Resta a los cuatro elementos contenidos en a, los de la misma posición en b.

__m128i _mm_srai_epi32(__mm128i a, int count);

Hace un desplazamiento a la derecha de los cuatro enteros de 32 bits contenidos en a. El desplazamiento será del numero de bits que indique el otro parámetro, *count*

__m128i _mm_store_si128(__m128i const *p, __m128i b)

Almacena en memoria los 128 bits que contiene b, a partir de la posición p. La dirección de p tiene que estar alineada en memoria.

__m128i _mm_storeu_si128(__m128i const *p, __m128i b)

Almacena en memoria los 128 bits que contiene b, a partir de la posición p. La dirección de p no tiene porque estar alineada en memoria, pero como pasa con el load, esta instrucción, aun más potente que la anterior, es menos eficiente que la instrucción __mm_store_si128.

__m128 _mm_cvtepi32_ps(_m128i a)

Convierte los cuatro enteros de 32 bits almacenados en a, a cuatro valores de 32 bits en punto flotante.

__m128i _mm_cvtpps_epi32(_m128 a)

Convierte los cuatro valores que hay en a de punto flotante a enteros.

__m128 _mm_mul_ps(_m128 a, _m128 b)

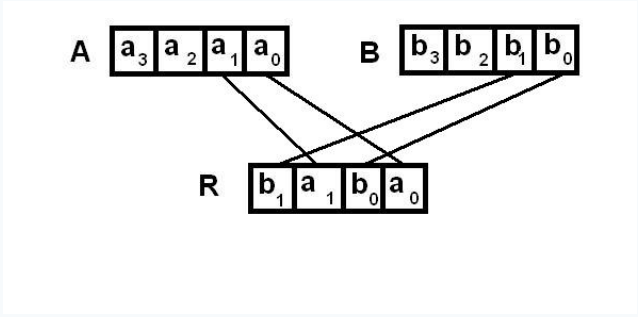
Multiplica los vectores a y b posición por posición. Todos los valores (tanto los originales como el resultado) estarán en punto flotante.

__m128i _mm_unpacklo_epi32(__m128i a, __m128i b)

Intercala los dos enteros que se encuentran en las posiciones menos significativas, con los enteros en las mismas posiciones de b.

r0 := a0 r1 := b0

r2 := a1 r3 := b1

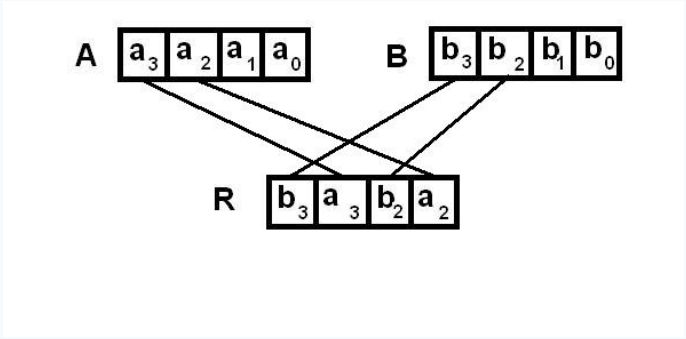


__m128i _mm_unpackhi_epi32(__m128i a, __m128i b)

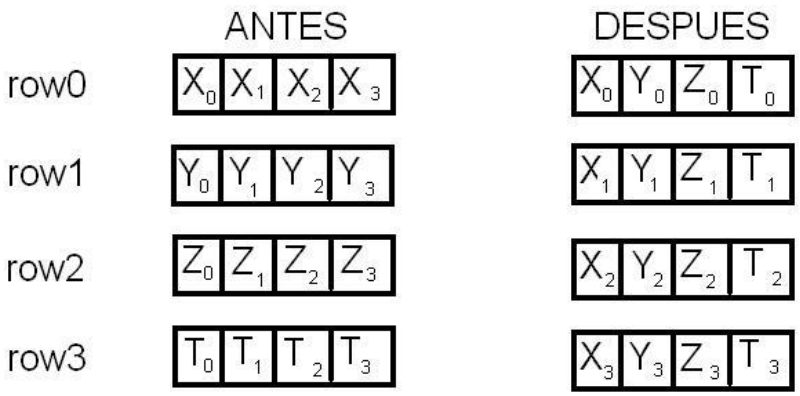
Intercala los dos enteros que se encuentran en las posiciones más significativas, con los enteros en las mismas posiciones de b.

r0 := a2 r1 := b2

r2 := a3 r3 := b3



A parte de las intrínsecas mencionadas, utilizamos una macro ya definida para transponer cuatro vectores. Esta es **_MM_TRANSPOSE4_PS(row0, row1, row2, row3)**. La imagen siguiente recoge la actuación de esta macro.



4. OPTIMIZACIÓN

Hemos partido de una implementación realizada por nosotros, pero sin ningún tipo de vectorización, y es la que usamos como base para posteriormente optimizarla introduciendo distintas técnicas de vectorización en el código.

La vectorización se basa, como su nombre indica, en el trabajo con vectores, que en nuestro caso serán de cuatro elementos, y en operar con dichos vectores, en vez de con cada elemento por separado, teniendo que hacer, por tanto, una cuarta parte de las operaciones deseadas. Esto se supone que conseguiría una mejora del 75% del tiempo total, pero no es así debido a que no todo el código es vectorizable, que las operaciones con vectores son más costosas que con elementos de tipos básicos y que aumenta el número de cargas y accesos a memoria, entre otras causas.

Para intentar esta optimización, hemos utilizado dos técnicas diferentes, en las que la vectorización por columnas coincide, pero en las que difiere la vectorización por filas, para intentar en una, hacer menos accesos a memoria y así ahorrarnos un valioso tiempo. Las dos técnicas empleadas en la vectorización por filas son la vectorización del propio filtro, y la vectorización por transposición. Esta última intenta usar la misma idea que la optimización por columnas.

A continuación exponemos las técnicas empleadas, con una breve descripción en pseudo-código de lo que hace cada una de ellas.

4.1 PSEUDOCODIGO PARA COLUMNAS

Paso 1: Calculo de los detalles de toda la fila 1, teniendo en cuenta el borde derecho.
Paso 2: Calculo de las aproximaciones de toda la fila 0, teniendo en cuenta el borde derecho.
Paso 3: Para cada columna, hasta el borde inferior
    Paso 3.1: Calculo del resto de las filas, calculando antes los detalles y luego las aproximaciones, incluyendo el borde derecho de cada fila.
Paso 4: Calculo del borde inferior dependiendo del tamaño de la imagen.

Con esta mejora, veremos que se consigue una gran optimización, ya que los accesos a memoria son mínimos, porque casi todas las cargas/accesos a memoria se hacen a bloques enteros de la misma, no tenemos que acceder a bloques distintos y después mezclarlos.

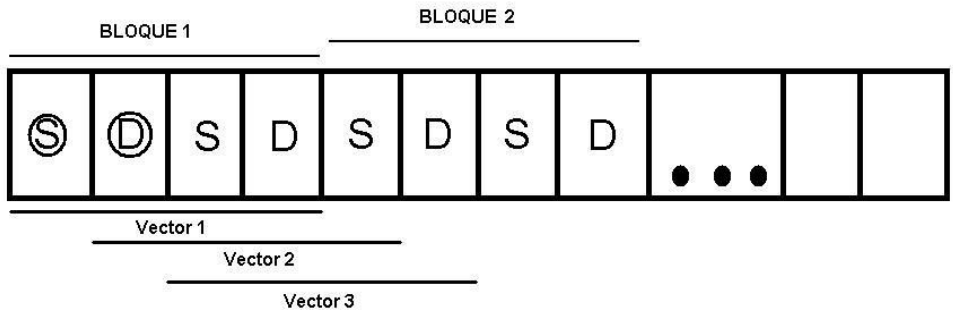
4.2 PSEUDOCODIGO PARA LAS FILAS

4.2.1 Vectorización del filtro

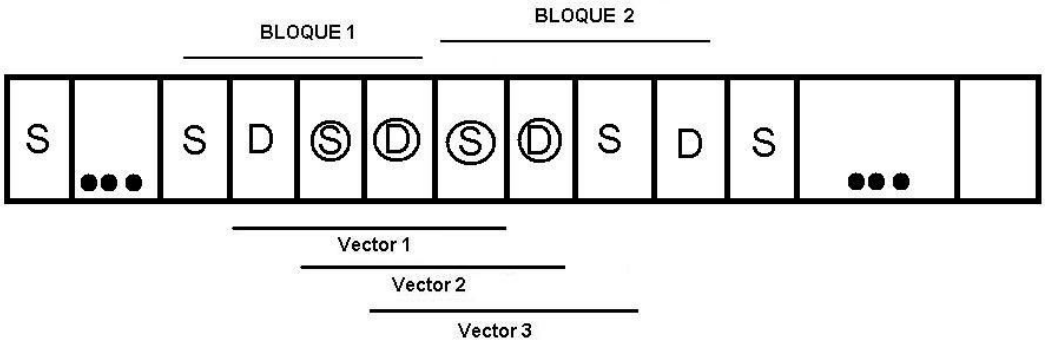
Paso 1: Para cada fila
- Paso 1.1: Calculo del borde izquierdo, primera detalle y primera aproximación
- Paso 1.2: Para el resto de la fila, excepto el borde derecho
  - Paso 1.2.1: Calculo de dos detalles y dos aproximaciones
- Paso 1.3: Calculo del borde derecho, que depende del tamaño de la imagen

Esta mejora tiene el problema de tener que acceder a datos de bloques de memoria distintos, con lo que es más costoso que si todos los datos a los que queremos acceder estuviesen en el mismo bloque de memoria.

En los siguientes gráficos se muestra lo que se hace. Los elementos redondeados son los que deseamos calcular. El primer dibujo representa el paso 1.1, y vemos que necesitamos tres vectores para calcular los dos elementos. El *vector 1* no da problemas, al coincidir con el *bloque 1*de memoria, pero observamos que el *vector 2* y el *vector 3* tienen datos en ambos bloques de memoria, con lo que su carga será mucho más costosa.



La segunda imagen, representa el paso 1.2.1, el cálculo de dentro del bucle interno. Podemos ver que en este caso ningún vector coincide con un bloque de memoria, consumiendo más tiempo en los accesos.



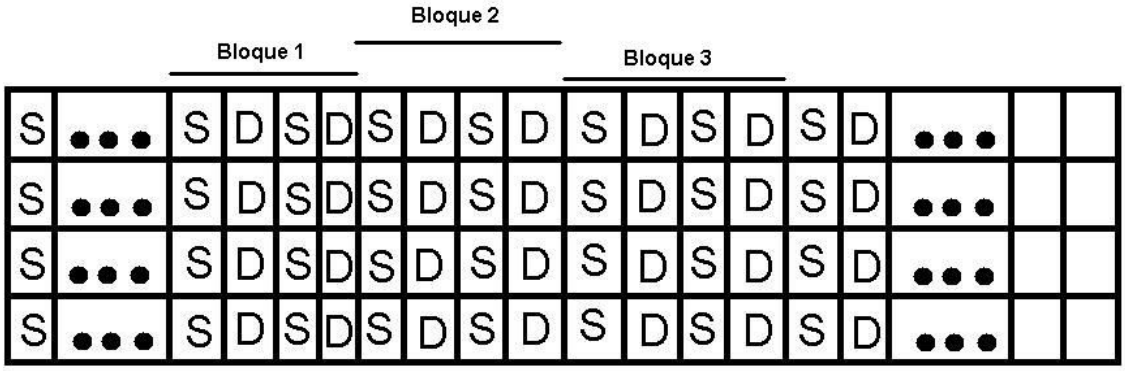
El borde derecho dependerá del tamaño de la imagen, ya que dependiendo de este, nos quedarán mas o menos datos que calcular, datos que pueden ser detalles o aproximaciones.

4.2.2 Vectorización por transposición

Paso 1: Cargamos ocho bloques en vectores, siendo las ocho primeras posiciones de las cuatro primeras filas, a cada submatriz 4x4 le llamaremos superbloque.
Paso 2: Transponemos los dos superbloques cargados en los vectores, y calculamos primero los detalles y las aproximaciones del primer superbloque.
Paso 3: Para el resto de la fila
    Paso 3.1: Cargamos el superbloque siguiente.
    Paso 3.2. Calculamos los resultados sobre el superbloque intermedio.
    Paso 3.3: Volvemos a transponer el primer superbloque y lo guardamos.
Paso 4: Calculamos los bloques derechos, cargando los datos que queden en un superbloque y transponiéndolo.
Paso 5: Volvemos a transponer los dos superbloques que quedan y los almacenamos.

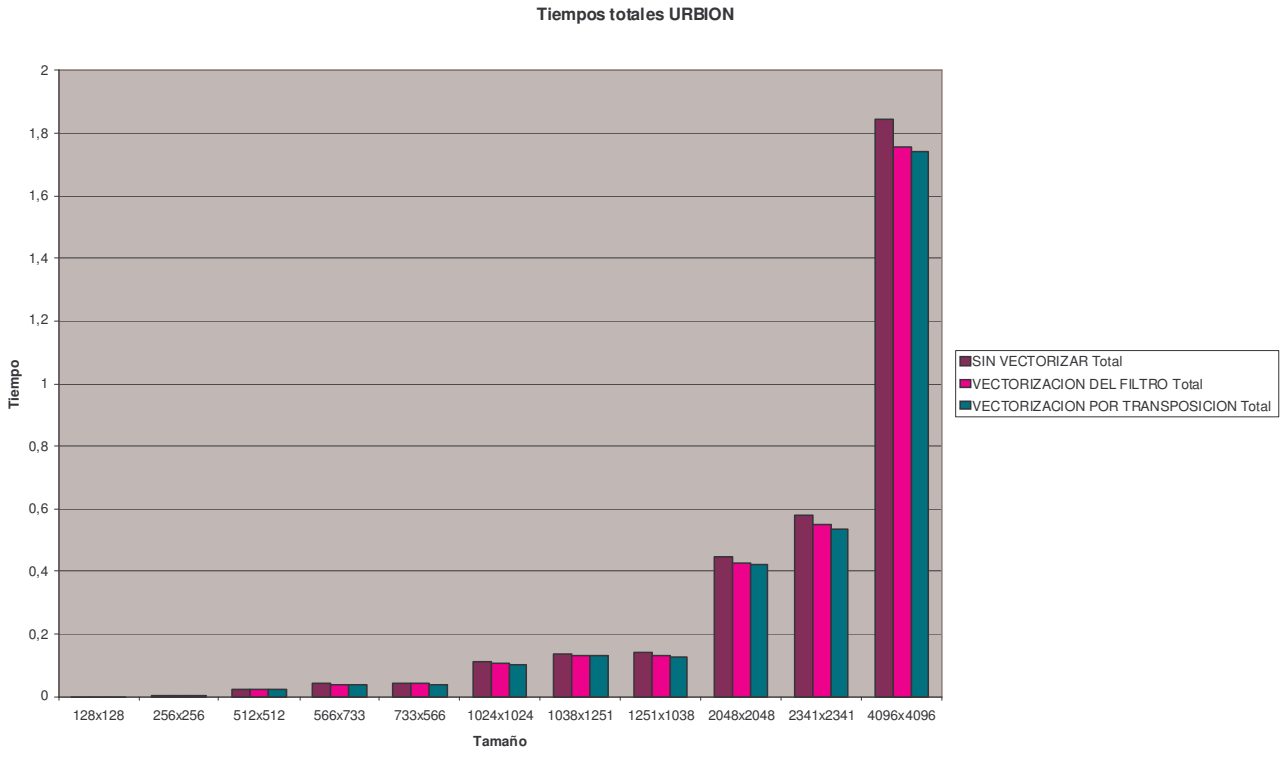
Con respecto a la vectorización del filtro, este método tiene la ventaja de acceder casi siempre a datos almacenados en el mismo bloque de memoria, por lo que las cargas/accesos disminuyen claramente con respecto a la mejora anterior. Esta es la misma idea que al vectorizar por columnas. Sin embargo, hay que tener en cuenta que esos datos hay que transponerlos, lo que también conlleva su tiempo, mientras que antes podíamos trabajar directamente con los datos cargados, sin necesidad de transponerlos.

La imagen siguiente muestra esquemáticamente el estado de las filas que estamos tratando. Los superbloques coinciden con los bloques de memoria, aunque en cada superbloque guardemos 16 valores, esto es, cuatro bloques (para la implementación, esto se traducirá en cuatro vectores), pero no guardaremos los bloques tal cual en los vectores, si no, transponiéndolos, pero esta transposición ya será una operación sobre vectores, mucho menos costosa que si fuera sobre memoria.



5.SPEED UP's

5.1 URBION TIEMPOS TOTALES

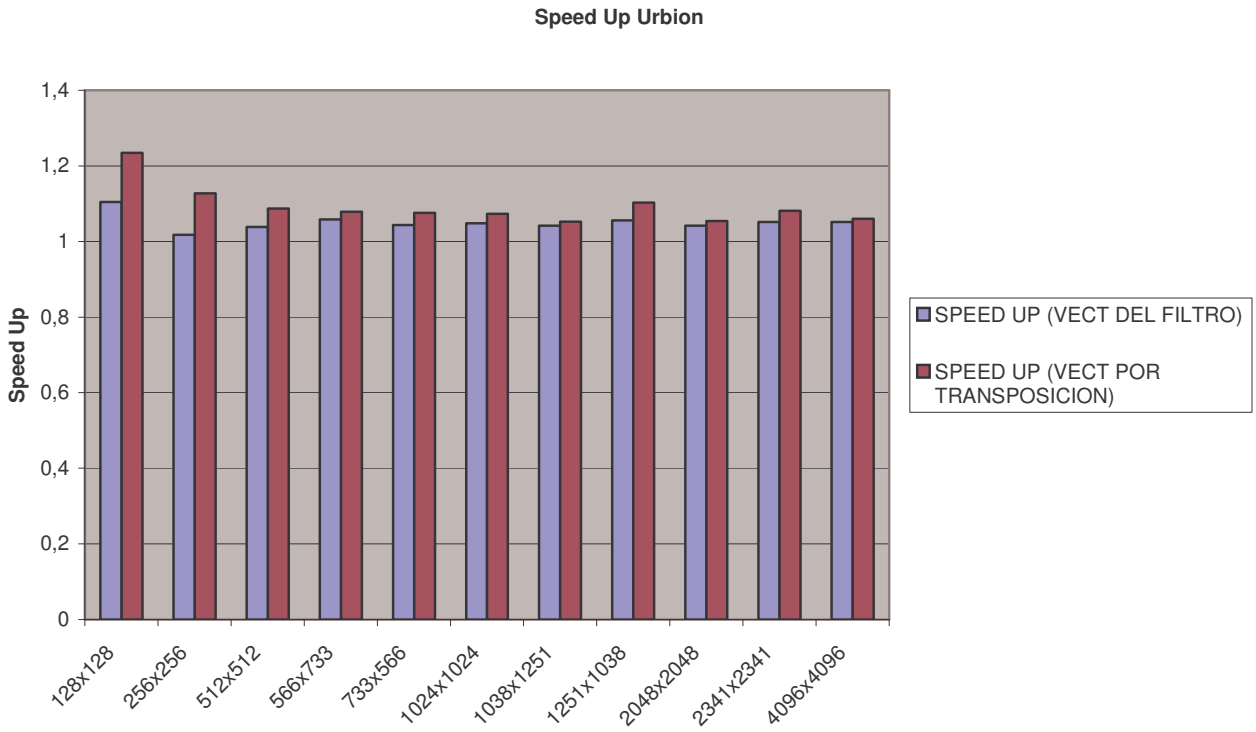


En este gráfico podemos comprobar el tiempo que tarda la maquina Urbión con tecnología PowerPC de Mac, en realizar la compresión de las imágenes con los tamaños abajo referenciados.

Podemos comprobar que los saltos mas significativos se dan con las imágenes cuadradas y potencias de dos. Como ocurre con los casos de 1024x1024, 2048x2048 y 4096x4096. En estos casos el tiempo aumenta considerablemente.

También podemos observar que los tiempos de las versiones vectorizadas son levemente inferiores a la versión "normal", la mejora no es muy significativa pero algo se mejora. En la siguiente transparencia vemos la mejora conseguida.

5.2 SPEED UP URBIÓN



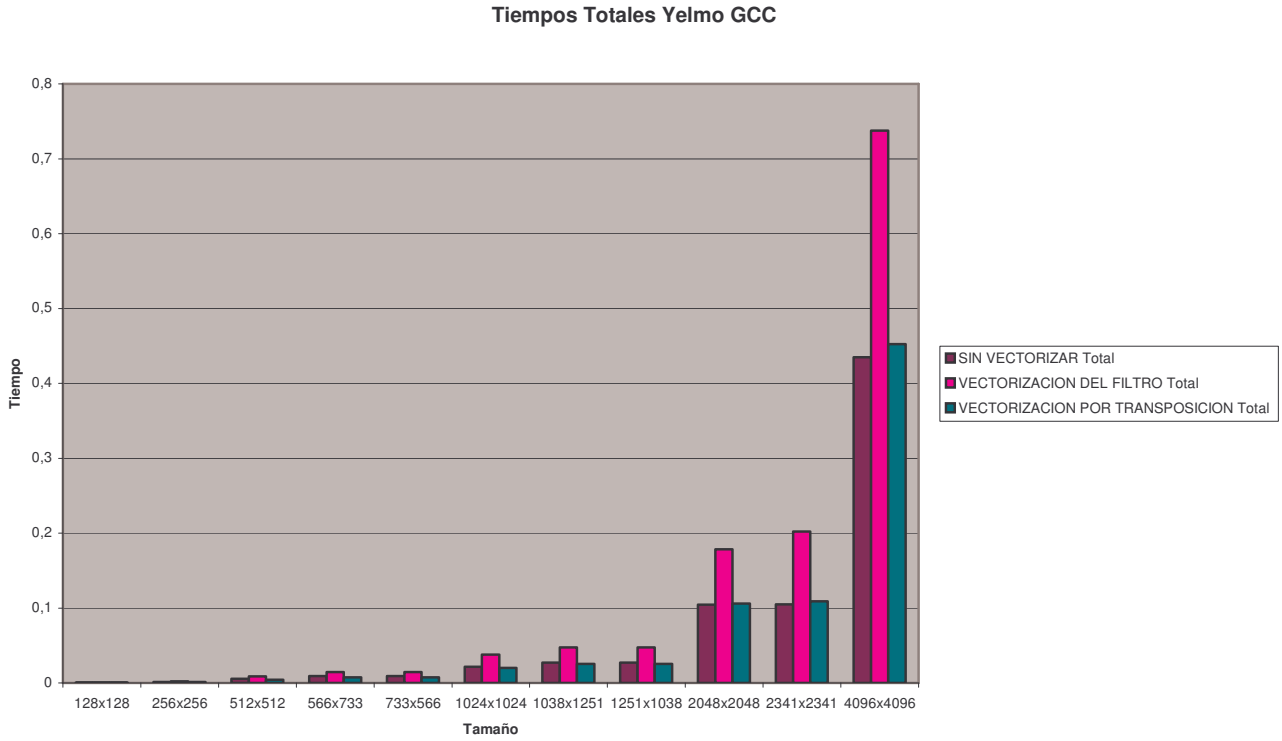
Como podemos observar aunque en todas las mejoras son superiores a uno no son excesivamente óptimas.

La vectorización del filtro ofrece una menor mejora que la versión en la que se vectoriza la transposición. Posteriormente veremos que lo que más influye sobre esta excasa mejora es la vectorización de la parte del filtro que trata las filas, puesto que la transposición de las columnas siempre ofrece altas mejoras.

El speed up más optimo se da con la imagen 128x128, q es la de menor tamaño, en ambas versiones. Cuando el tamaño aumenta la mejora va empeorando, así podemos comprobarlo con la imagen de 4096x4096, en la que la mejora es la menor, aunque no la minima aunque la tendencia es de estabilización por levemente por encima del uno.

La mejora con la versión de la transposición es más fluctuante, sin embargo en la versión del filtro la mejora se va estabilizando aunque nunca supera el speed up de la otra versión.

5.3 YELMO GCC - TIEMPOS TOTALES

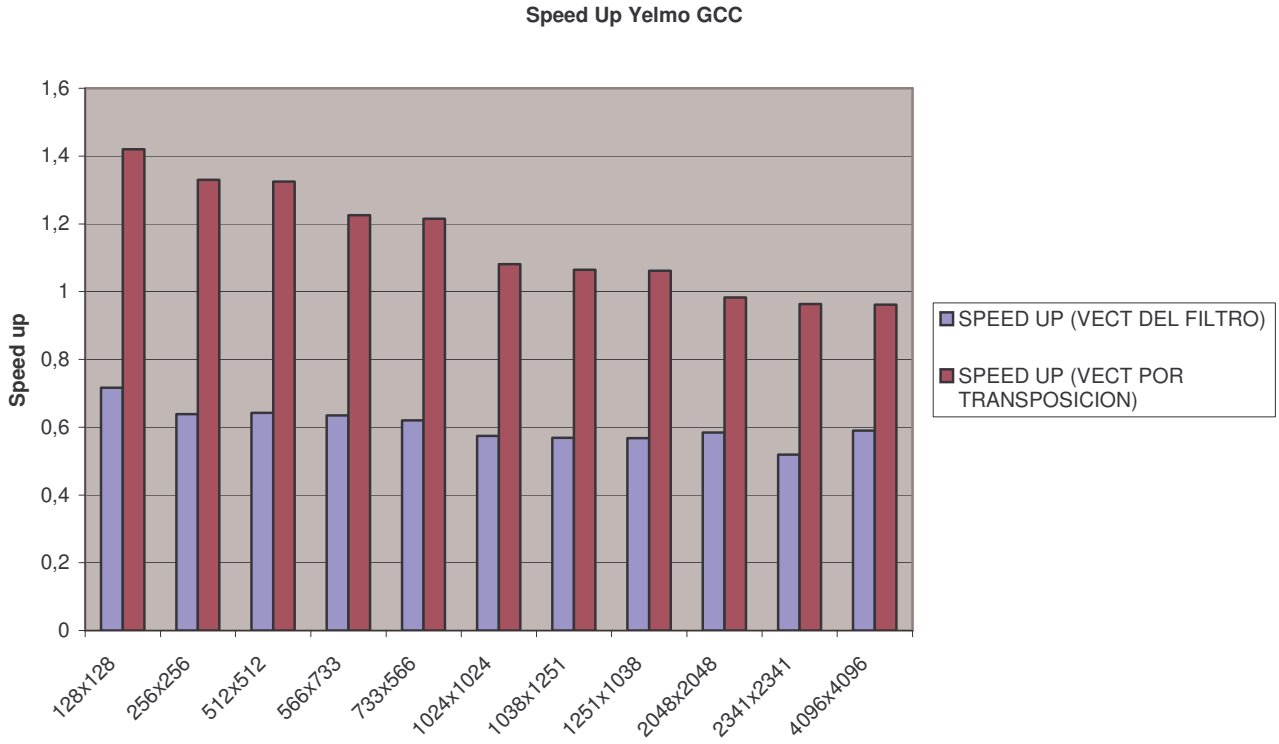


A primera vista dos hechos llaman nuestra atención, los tiempos son mucho menores que en Urbión, y que la versión del filtro no optimiza, es más da tiempos mucho mayores que la versión sin vectorizar.

Los tiempos en la maquina Yelmo, con las versiones compiladas con el compilador gcc en su versión 4.0 (todavía β) no superan 0.5 seg. de tiempo, excepto la versión que vectoriza el filtro para la imagen 4096x4096, que casi dobla el tiempo de la versión sin vectorizar, aunque no llega ni a 1seg. Aunque comentamos las significativas diferencias de tiempos entre ambas máquinas, no es debido a la diferente arquitectura que poseen, puesto que los procesadores utilizados en la toma de resultados no son equivalentes.

Lo que tenemos que resaltar es la “poca” optimización que nos proporciona la versión que vectoriza el filtro. También resaltamos que aunque en las imágenes más pequeñas, como veremos en el speed up, la versión que vectoriza por transposición si obtiene mejora, con las imágenes grandes (4096x4096) ya no existe esa mejora.

5.4 SPEED UP YELMO GCC

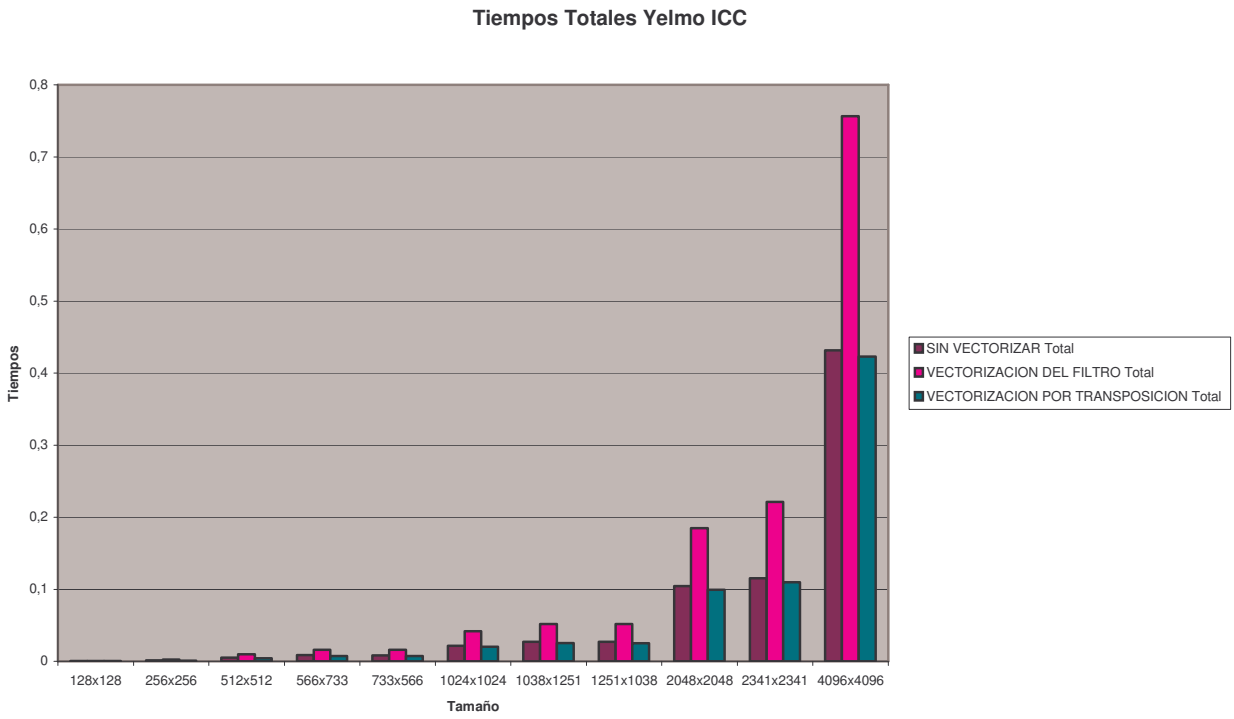


Primeramente destacar que la versión que vectoriza el filtro no ofrece ninguna mejora, es más en todos los casos emplea casi el doble que la versión sin vectorizar en realizar la compresión.

En la versión que vectorizada por transposición al principio la mejora se aproxima al 1.5, pero esta mejora se va perdiendo cuando aumenta el tamaño de la imagen, así podemos ver que con las tres ultimas imágenes la versión vectorizada tarda un poco mas que la versión sin vectorizar.

En comparación con Urbión, en el aspecto de la mejora, la primera maquina que hemos visto, tiene una mejora menor al principio y mejor al final, pero es mas constante. Sin embargo Yelmo posee mas mejora con las imágenes pequeñas, pero su mejora se decrementa mas rápidamente.

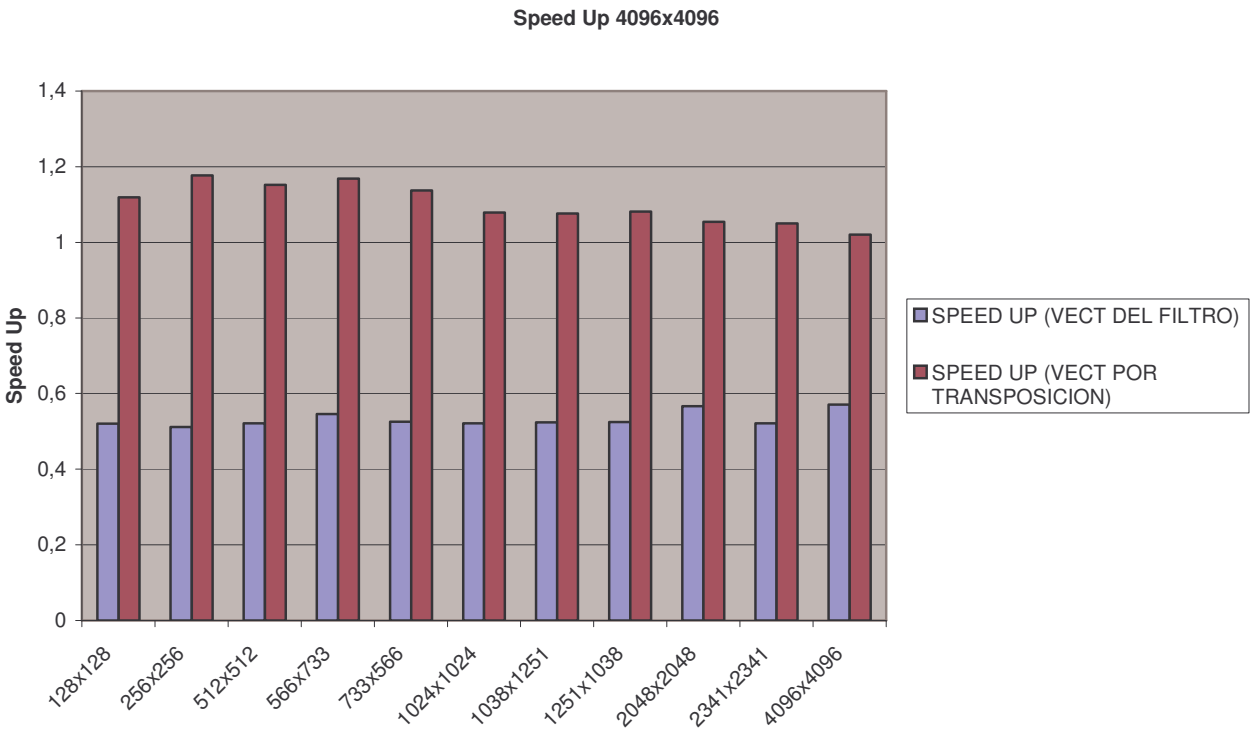
5.5 YELMO ICC TIEMPOS TOTALES



Los tiempos son similares a los anteriores, quizás un poco superiores, puesto que lo único que los diferencia es el compilador, en este caso el utilizado es el ICC, propio para la arquitectura Intel.

Notamos las mismas diferencias, la poca mejora que nos facilita la versión que vectoriza el filtro, y el descenso de la mejora en la otra versión.

5.6 SPEED UP YELMO ICC



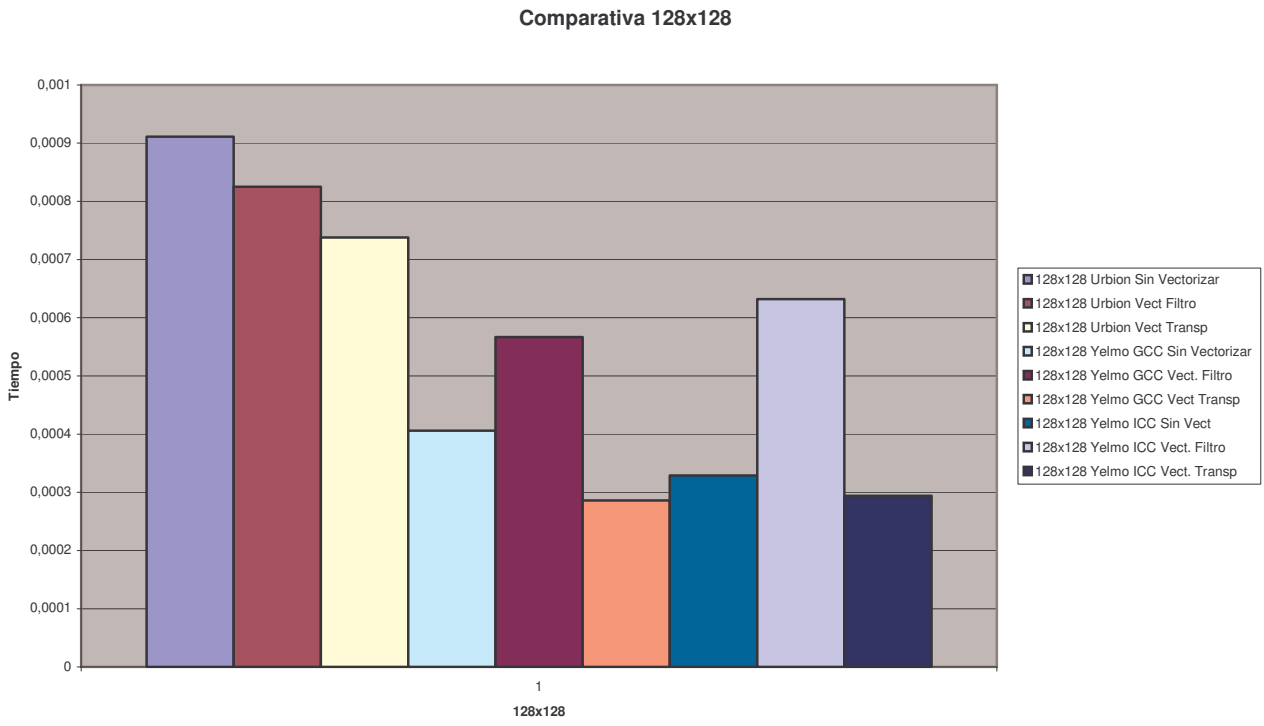
La mejora de la versión que vectoriza el filtro es mucho menor que con el otro compilador, para la misma máquina. En pocos casos supera el 0.5 de speed up, no como con la compilación del gcc, que siempre lo supera.

Para la otra versión podemos destacar, en ningún caso supera el 1.2 de speed up, pero tampoco desciende de 1, no como los resultados obtenidos con el otro compilador, que para las imágenes grandes (2048x2048, 2341x2341 y 4096x4096) no lo supera, es decir, el tiempo de la versión vectorizada tarda mas tiempo que la que no lo esta. La tendencia es a llegar a uno, pero en ninguno de los casos testeados la versión vectorizada y compilada con el ICC tarda más que la versión sin vectorizar.

Aun así el mejor speed up para imágenes grandes, 4096x4096, se obtiene con la maquina Urbión, aunque su tiempo total de ejecución es mayor que con la máquina Yelmo, la mejora con la vectorización es mejor.

Entre ambas versiones vectorizadas la que mejores resultados da es la que vectoriza la trasposición. Aunque ambos utilizan el mismo proceso para calcular las columnas, lo que les difiere son las filas, y los mejores resultados se consiguen realizando la trasposicion de las filas, en vez de vectorizar la fila.

5.7 IMAGEN 128x128



Repetimos los hechos ya comentados en los puntos anteriores:

1.- La asociación versión-arquitectura que más tarda es la que no está vectorizada en la máquina Urbión.

2.- Para la máquina Urbión ambas versiones vectorizadas en esta imagen pequeña si que ofrecen mejora.

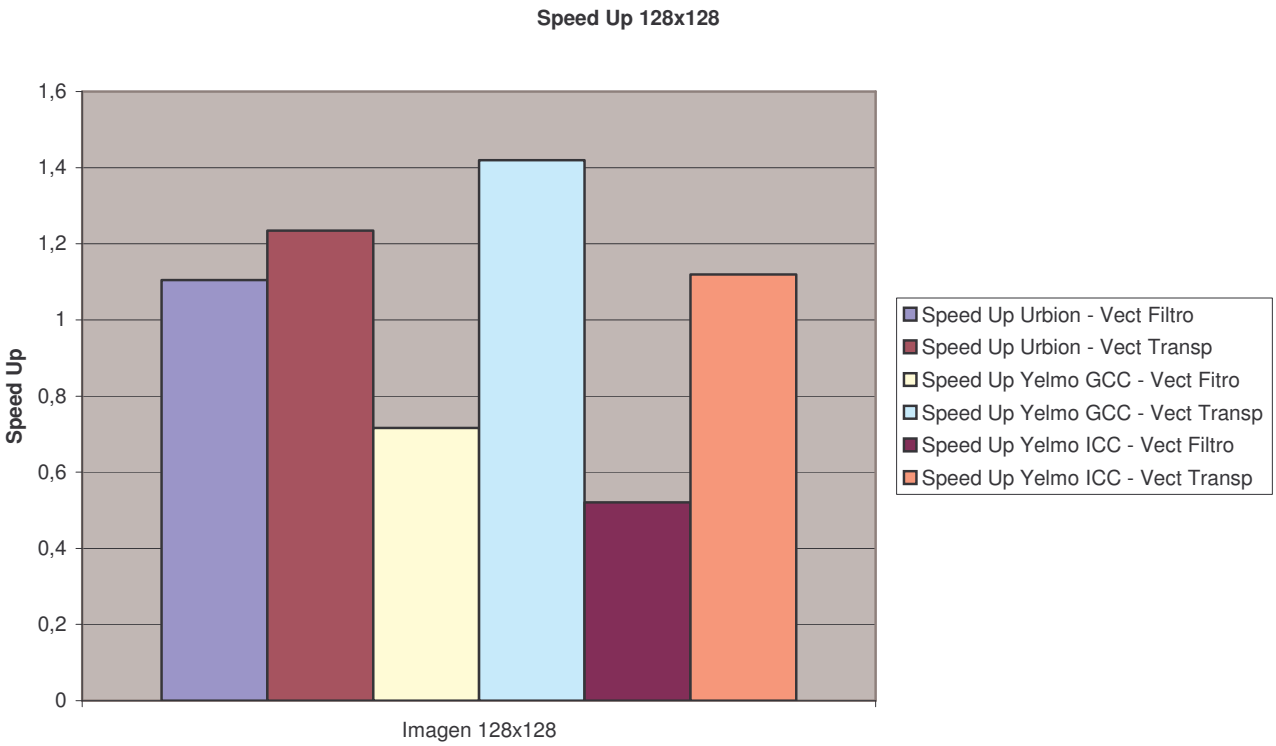
3.- Siendo la que menor tiempo emplea la versión que vectoriza trasponiendo las filas, de estas dos.

4.- En la máquina Yelmo la versión que vectoriza el filtro de la wavelet causa tiempos mucho mayores que la versión que no vectorizada.

5.- Con la compilación GCC 4.0 es la que proporciona menores tiempos en las ejecuciones en la máquina Yelmo, excepto para la versión que vectoriza el filtro.

6.- La mejora para con la compilación GCC 4.0 en Yelmo para esta imagen pequeña es mejor que la compilación con ICC.

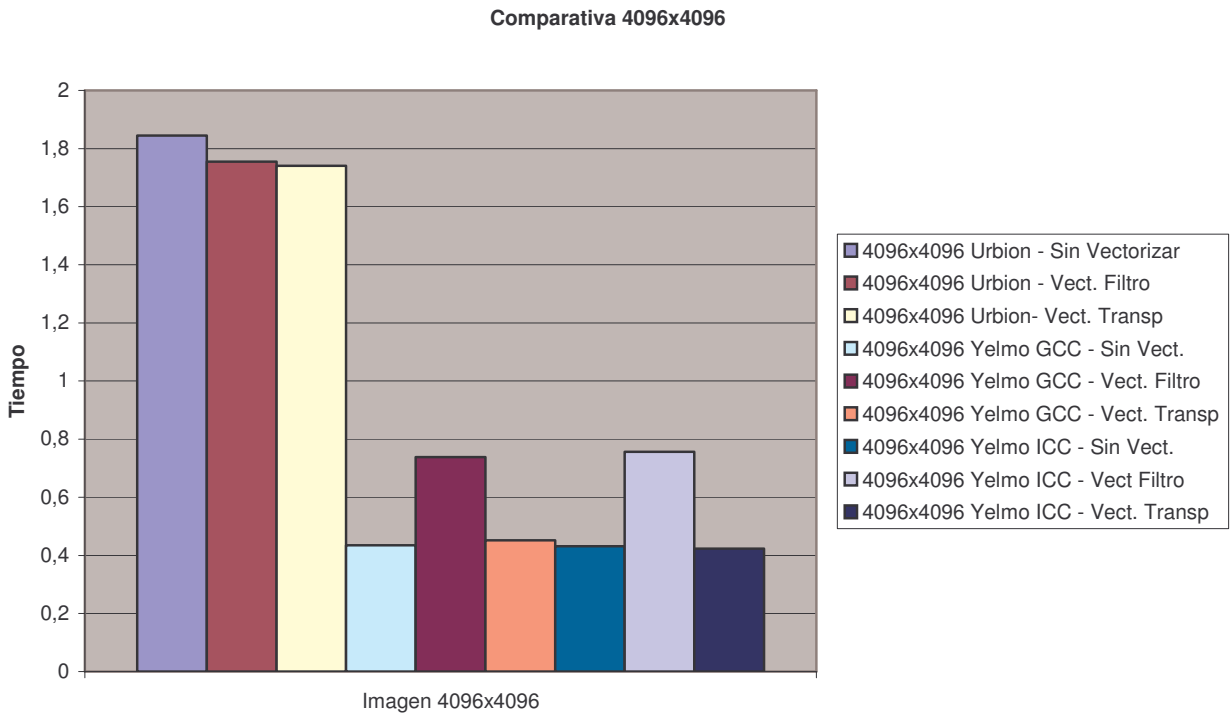
5.8 SPEED UP 128x128



La mejor mejora para esta imagen es que vectoriza por trasposición con la compilación GCC 4.0, en la maquina Yelmo.

Las únicas inferiores a cero son aquellas que vectorizan el filtro en la maquina Yelmo, aunque la imagen comprimida es la menor de las tratadas. Ya vimos antes que en ningún caso ofrecen mejora en esta máquina.

5.9 IMAGEN 4096x4096



Repetimos los hechos ya comentados en los puntos anteriores:

1.- La asociación versión-arquitectura que más tarda es la que no está vectorizada en la máquina Urbión, igual que para imágenes pequeñas.

2.- Para la máquina Urbión ambas versiones vectorizadas en esta imagen tambien ofrecen mejora.

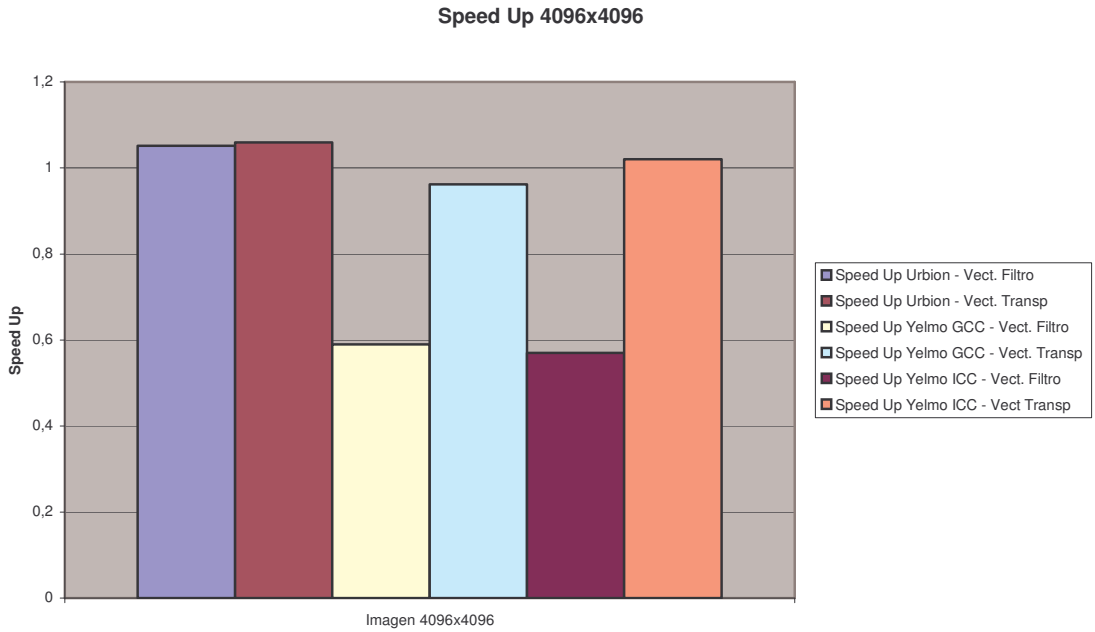
3.- Siendo la que menor tiempo emplea la versión que vectoriza trasponiendo las filas, de estas dos.

4.- En la máquina Yelmo la versión que vectoriza el filtro de la wavelet causa tiempos mucho mayores que la versión que no vectorizada. Aunque para esta imagen la vectorizacion por transposición tampoco ofrece mejora.

5.- Con la compilación GCC 4.0 es la que proporciona menores tiempos en las ejecuciones en la máquina Yelmo, excepto para la versión que vectoriza el filtro.

6.- La mejora para con la compilación GCC 4.0 en Yelmo para esta imagen grande es mejor que la compilación con ICC.

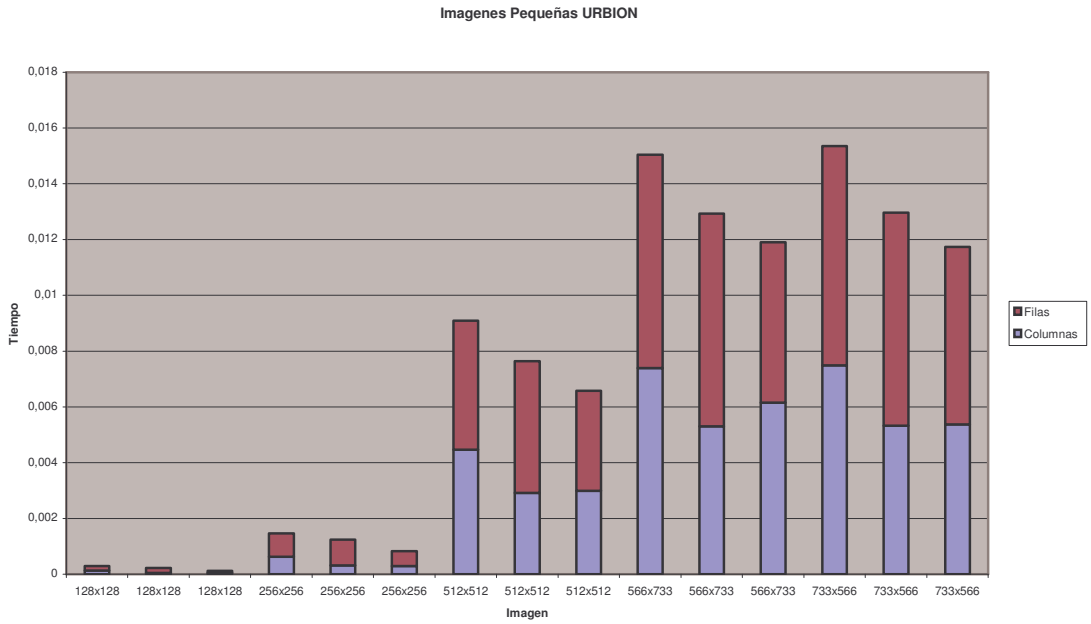
5.10 SPEED UP 4096x4096

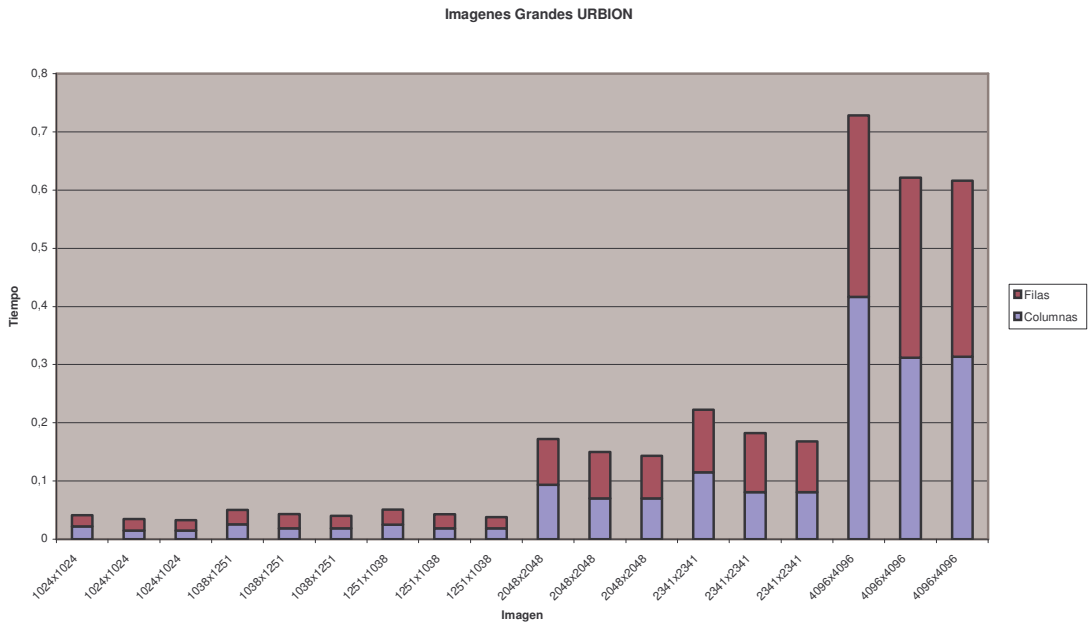


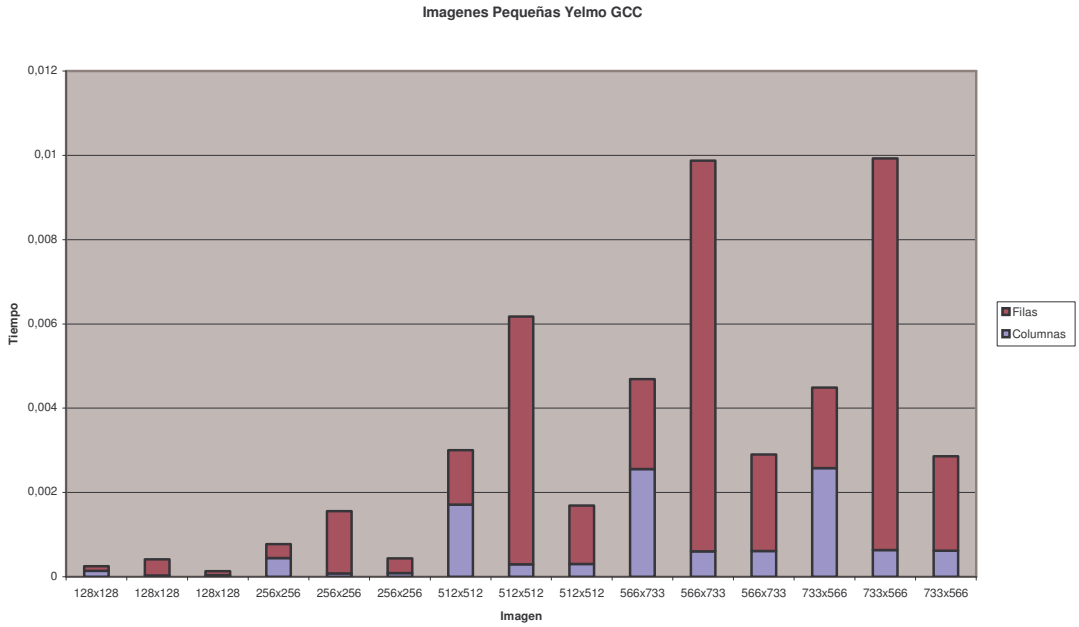
La mejor mejora para esta imagen es la vectorización por trasposición en la maquina Urbión, aunque las mejoras son significativamente peores, puesto que son muy próximas a uno, que las mejoras obtenidas con la imagen pequeña.

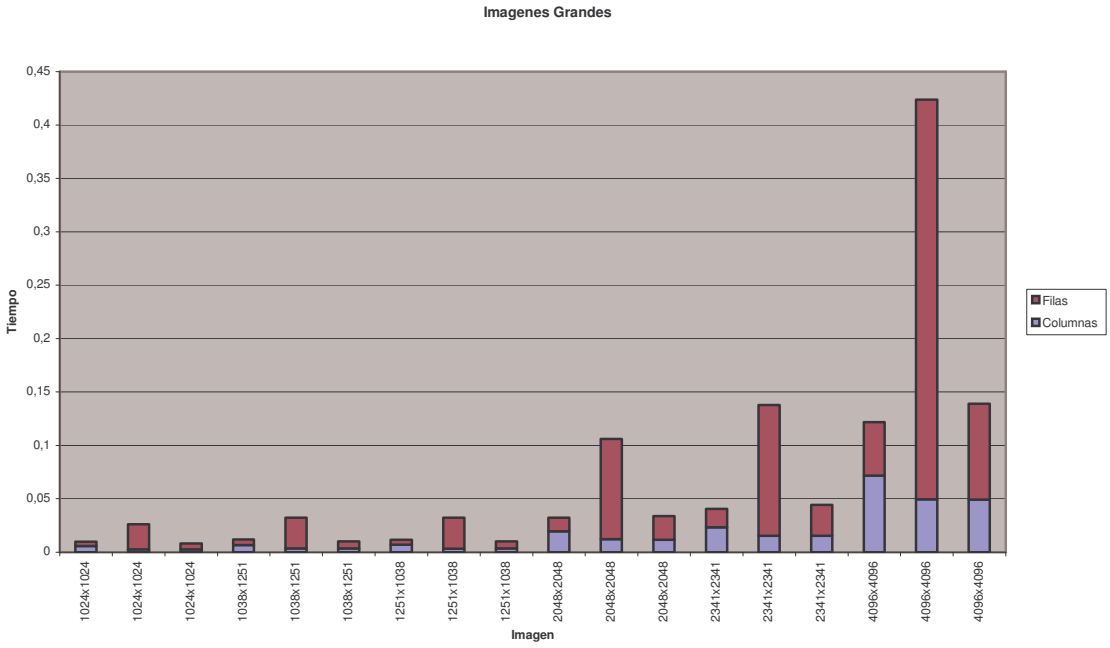
Llama la atención el hecho de que solo tres de las seis versiones utilizadas ofrezcan mejora para la imagen más grande de las tratadas, y que dos de estas tres sean las que se aplican a la arquitectura de la máquina Urbión. Solo una de las cuatro versiones que utiliza el repertorio de instrucciones que proporciona intel para vectorizar ofrece mejora, y hemos de notificar que esta mejora es menor que las otras dos que nos proporcionan las versiones que utilizan el repertorio de instrucciones de la máquina con arquitectura PowerPC.

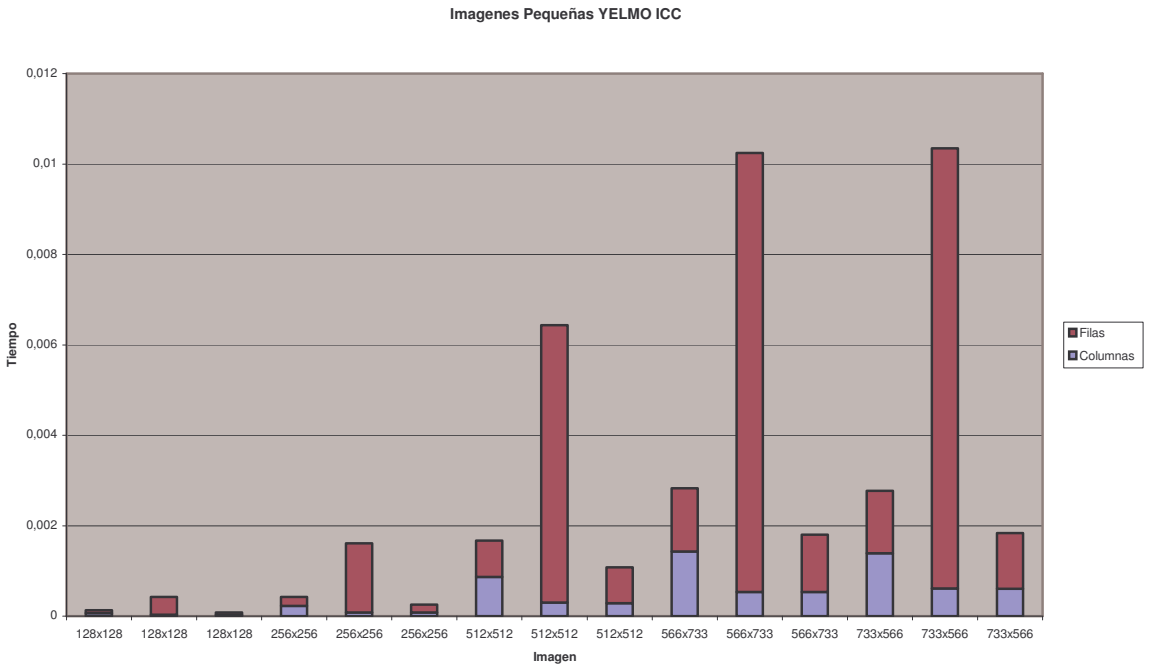
5.11 COMPARATIVAS FILAS-COLUMNAS

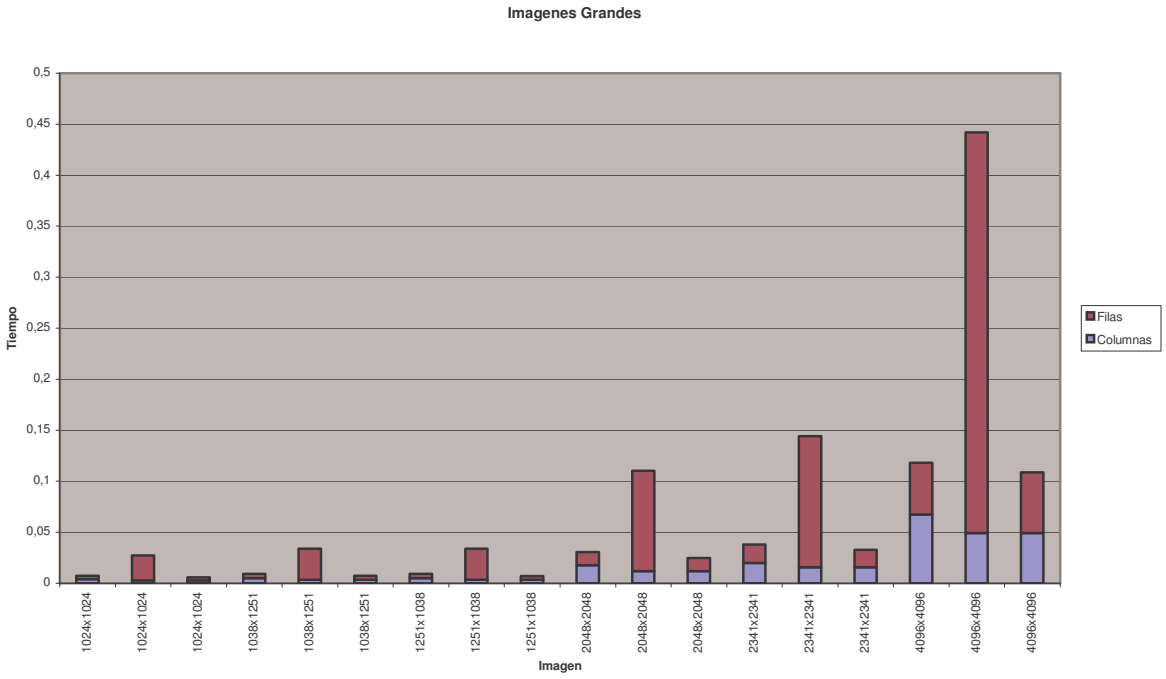





Imagenes Pequeñas Yelmo GCC
0,012
0,01
0,008
0,006
0,004
0,002
0
Tiempo
128x128
128x128
128x128
256x256
256x256
256x256
512x512
512x512
512x512
566x733
566x733
566x733
733x566
733x566
733x566
Imagen
Filas
Columnas


Imagenes Grandes
0,45
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
Tiempo
1024x1024
1024x1024
1024x1024
1038x1251
1038x1251
1038x1251
1251x1038
1251x1038
1251x1038
2048x2048
2048x2048
2048x2048
2341x2341
2341x2341
2341x2341
4096x4096
4096x4096
4096x4096
Imagen
Filas
Columnas





Puesto que utilizamos el mismo algoritmo para implementar las columnas en ambas versiones, y este es bastante optimo, lo que se debería de optimizar es la parte de las filas, que es la que introduce los mayores tiempos en el resultado total.

Referencias

www.wikipedia.org

http://coco.ccu.uniovi.es/immed/compresion/descripcion/fundamentos/fundamentos.htm

JPEG2000. Un nuevo estándar en comprensión de imágenes y video digital.
Rafael Redondo y Gabriel Cristóbal. Instituto de Óptica (CSIC), Serrano 121, 28006 Madrid

Intel® C++ Compiler for Linux Systems User's Guide
(c) Intel Corporation 1996-2004

Wavelet Compression and the JPEG2000 Standard.
Majid Rabbani, Eastman Kodad Research Laboratories
Majid.rabbani@kodak.com